Name: *Marius Juston*
NetID: *mjuston2*
Section: *AL2*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.219294ms*	*0.818139ms*	*1.037433ms*	*0.86*
1000	*2.07119ms*	*8.07871ms*	*10.1499ms*	*0.886*
10000	*20.4125ms*	*78.6866ms*	*99.0991ms*	*0.8714*

1. **Optimization 1: *Tiled Shared Memory Convolution + Constant kernel memory***

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

    For the first optimization, I decided to implement the Tiled shared memory convolution with, in addition, placing the kernel in constant memory. I decided to choose this optimization technique because it was the next easiest step from the current non-optimized version of the kernel.

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

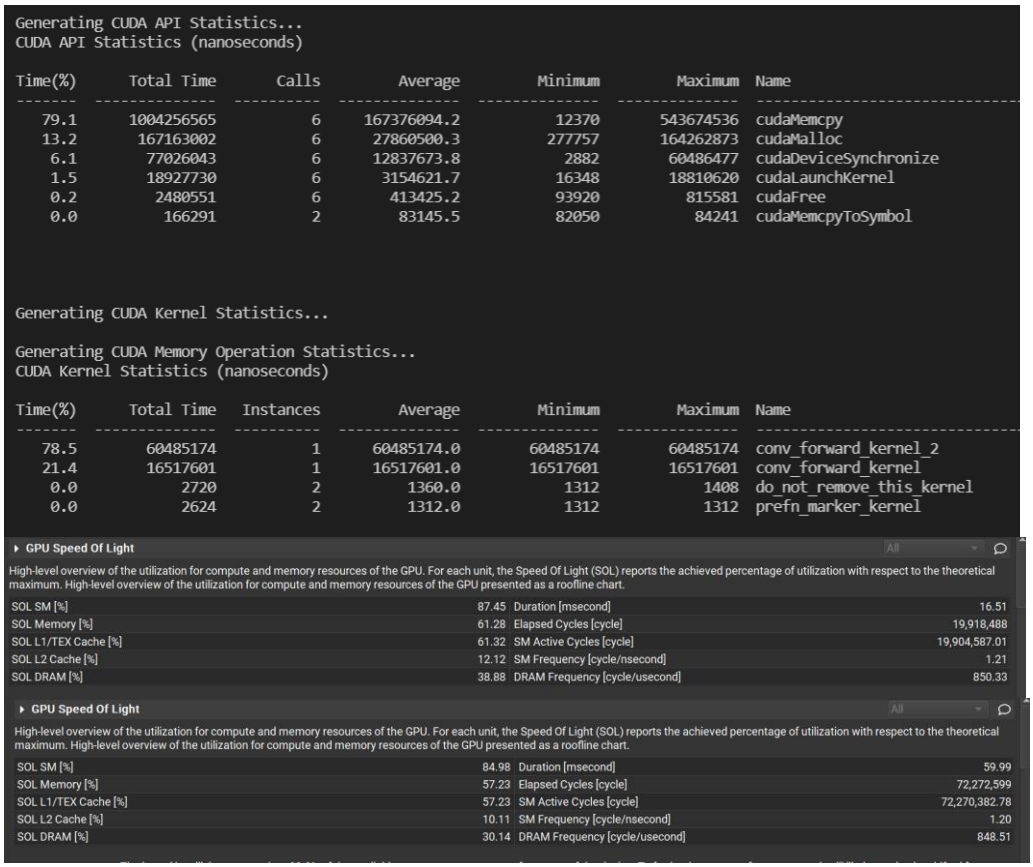
    This optimization works by first placing the kernel into constant memory. Then each thread loads into memory one, two, or three elements of the input matrix into a shared memory array, from that it then performs the convolution. A thread can load up to three elements because for the shared memory we need to store the edges required for the edge convolutions. Since this is in 2D we need some threads to handle loading data from the right and the bottom of the input matrix as well as another group for the bottom right corner. I believe that the optimization would increase the performance of the forward convolution because we were making use of shared memory to improve the reuse performance and thus reduce the number of read and write cycles to global memory. In turn, this optimization works well with the constant kernel memory since that aims to produce a similar effect.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.174969	0.623188	0.798157	*0.86*
1000	1.68267	6.12279	7.80546	*0.886*
10000	16.5975	60.8619	77.4594	*0.8714*

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

This optimization did indeed successfully improve the performance. Thanks to the shorter waiting times for the reads and writes to global memory the kernel was able to execute faster. This can be demonstrated in the OP times compared to the baseline where the times are consistently smaller. This can be noticed in the duration of 16.51ms compared to the baseline 20.32ms for Layer 1 and 59.99ms vs 81.79ms for Layer 2. In addition, this optimization has a little better utilization of the SM with the SOL SM % for the baseline layer being 81.51% and 78.82% respectively while this optimized version is 87.45% and 84.98.



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   79.1      1004256565           6     167376094.2           12370       543674536  cudaMemcpy
   13.2       167163002           6      27860500.3          277757       164262873  cudaMalloc
    6.1        77026043           6      12837673.8            2882        60486477  cudaDeviceSynchronize
    1.5        18927730           6       3154621.7           16348        18810620  cudaLaunchKernel
    0.2         2480551           6        413425.2           93920          815581  cudaFree
    0.0          166291           2         83145.5           82050           84241  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   78.5        60485174           1      60485174.0        60485174        60485174  conv_forward_kernel_2
   21.4        16517601           1      16517601.0        16517601        16517601  conv_forward_kernel
    0.0            2720           2          1360.0            1312            1408  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1312            1312  prefn_marker_kernel
```

▸ 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 [%]	87.45	Duration [msecond]	16.51
SOL Memory [%]	61.28	Elapsed Cycles [cycle]	19,918,488
SOL L1/TEX Cache [%]	61.32	SM Active Cycles [cycle]	19,904,587.01
SOL L2 Cache [%]	12.12	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	38.88	DRAM Frequency [cycle/usecond]	850.33

▸ 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 [%]	84.98	Duration [msecond]	59.99
SOL Memory [%]	57.23	Elapsed Cycles [cycle]	72,272,599
SOL L1/TEX Cache [%]	57.23	SM Active Cycles [cycle]	72,270,382.78
SOL L2 Cache [%]	10.11	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	30.14	DRAM Frequency [cycle/usecond]	848.51

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

Lecture notes

2. **Optimization 2: Kernel fusion for unrolling and matrix-multiplication + Constant Kernel memory *+ Thread block optimization***
   a. Which optimization did you choose to implement and why did you choose that optimization technique?

   For this optimization, I decided to implement the kernel fusion for the matrix multiplication and unrolling as well as the utilization of shared memory for the matrix multiplication for the input variable and the use of constant memory for the kernel. I decided to make use of matrix unrolling because this format should help with the global memory bandwidth and possibly help speed up larger convolutions.

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

   This optimization works by converting the traditional way of implementing convolutions, by sliding a kernel around the input, into a matrix multiplication operation. Though to implement this matrix multiplication the input matrix needs to be "unrolled" to format the convolutions of the matrix as the columns of the matrix. To unroll, into shared memory, each thread takes care to handle its part of remapping the input matrix into shared memory to be used later on. Once all the cached data is set a traditional dot product is used to compute the convolution out. I believed that this optimization would increase the performance of the forward convolution; however, I expect it to provide more benefits on the larger matrix, Layer 2 because this method helps with memory bandwidth issues and thus is more efficient on larger operations. This optimization also works well with the constant memory optimization and is an addon to the Shared memory matrix multiplication and input matrix unrolling. For Layer 1 a tile size of 16 was utilized while Layer 2 had a tile size of 32.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.393132	0.527104	0.920236	*0.86*
1000	3.79919	5.31126	9.11045	*0.886*
10000	37.782	53.5231	91.3051	*0.8714*

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

This optimization was able to partially improve the performance of the program. As expected, compared from optimization 1 with the traditional convolution layer, the Op Time for Layer 1 is slower while the larger matrix multiplication, Layer 2, which can make better use of the memory bandwidth improvements, sees performance improvements. For Layer 1 the unrolling operation might be causing more computation slowdown compared to its actual benefits. From the SOL Memory %, we are also able to see that this optimization can reduce the memory utilization by about 20-30% compared to Optimization 1. For the final implementation, the utilization of the shared tile convolution method is used for Layer 1 and this unrolled tiled memory is used for Layer 2.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   77.5       981139368           6     163523228.0           11412       514173930  cudaMemcpy
   13.8       174513861           6      29085643.5          272259       171378107  cudaMalloc
    7.3        92650224           6      15441704.0            2792        54076342  cudaDeviceSynchronize
    1.2        15439307           6       2573217.8           14274        15323045  cudaLaunchKernel
    0.2         2366450           6        394408.3           78370          756746  cudaFree
    0.0          349534           2        174767.0          171932          177602  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   58.4        54075150           1      54075150.0        54075150        54075150  conv_forward_kernel_2
   41.6        38552665           1      38552665.0        38552665        38552665  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2688           2          1344.0            1312            1376  prefn_marker_kernel
```

GPU Speed Of Light — All

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 [%]	83.91	Duration [msecond]	37.48
SOL Memory [%]	30.80	Elapsed Cycles [cycle]	45,235,391
SOL L1/TEX Cache [%]	30.80	SM Active Cycles [cycle]	45,228,874.89
SOL L2 Cache [%]	6.26	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	5.39	DRAM Frequency [cycle/usecond]	850.17

GPU Speed Of Light — All

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 [%]	68.59	Duration [msecond]	53.58
SOL Memory [%]	32.24	Elapsed Cycles [cycle]	65,807,507
SOL L1/TEX Cache [%]	32.59	SM Active Cycles [cycle]	65,093,676.56
SOL L2 Cache [%]	2.60	SM Frequency [cycle/nsecond]	1.23
SOL DRAM [%]	2.79	DRAM Frequency [cycle/usecond]	865.18

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

Lectures 13, NVIDIA Blogs

3. **Optimization 3:** ***FP16 Optimization + Thread block optimization + Unrolled Matrix***

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

This optimization is the utilization of FP16 operations in the kernel, this is used in the unrolled matrix operation in Optimization 2. Again Layer 1 has a tile size of 16 and Layer 2 has a tile size of 32. I chose this optimization technique because it is supposed to further reduce the time to compute floating-point operations since it converts the usual 32-bit floating points into 16-bit floating points making it more memory efficient and faster to calculate.

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

This optimization works by converting the float data types to __half or __half2. These are floating-point data types with only 16 bits instead of float's 32 bits. By converting the size of float, you can both save memory size, since the variable is twice as small and, since there are fewer bits, performing operations such as addition and multiplications are normally much faster. I believe that this should increase the performance of the forward convolution. This optimization should also be able to be used by all the other optimizations to get a little bit more performance boost. However, the cost of this performance boost comes in the shape of a drop inaccuracy. The precision of FP16 is much smaller than its 32-bit counterpart and thus accumulates error more.

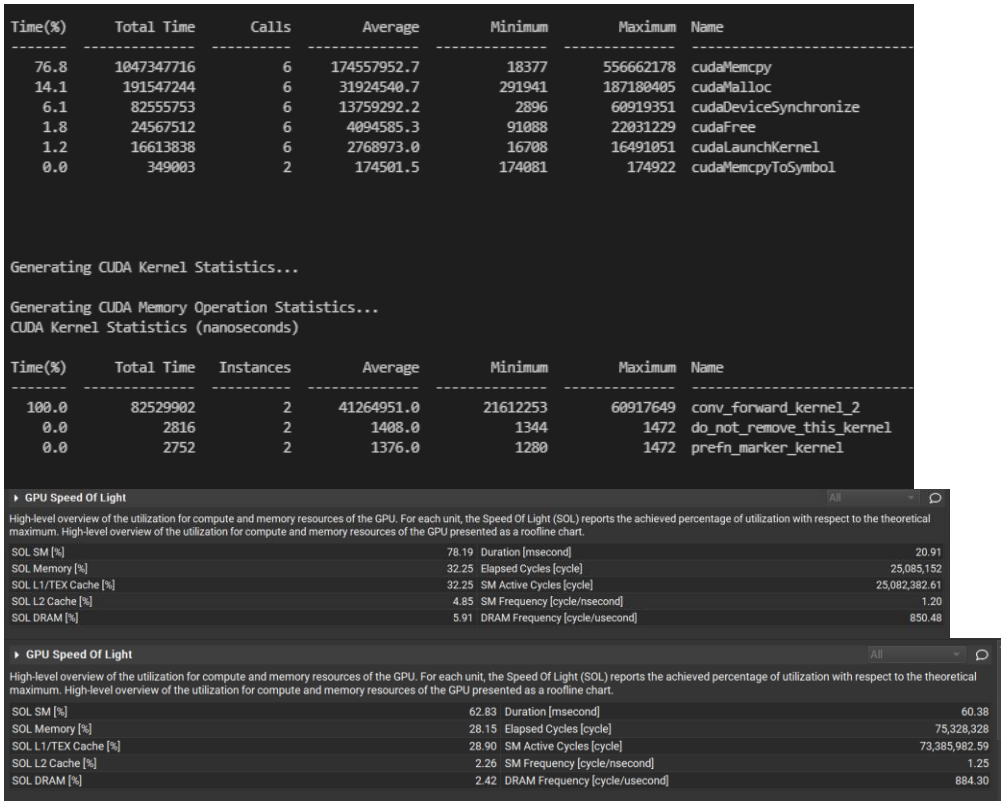
    c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.223222	0.602509	0.825731	0.84
1000	2.12114	6.01286	8.134	0.87
10000	20.9049	60.0765	80.9814	0.8636

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

Sadly, it seems like the implementation of this performance did not work. Though I believe that it might be due to how I implemented the computation and conversion. I did not see much/good documentation on how to implement FP16 kernels and how to convert the floating-point arrays to half2/half format and so if implemented a different way you might be able to achieve the expected results. The FP16 Layer times for Layer 2 are slightly slower than the base Optimization 2 with

matrix unrolling; however, the FP16 with Layer 1 seems to have increased the performance of the kernel. Though, strangely, the amount of memory usage has changed very little. The SOL Memory % of Layer 1 with FP16 was 32.25% compared to without it 30.80%, I believe that this number should have been smaller. However, for layer 2 a small memory reduction was noticed with 28.15% being used instead of 32.24%. It is also strange that the SOL SM% decreased for Layer 1, from 83.91% to 78.19% though it might be attributed to the increased memory usage.



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   76.8      1047347716           6     174557952.7           18377       556662178  cudaMemcpy
   14.1       191547244           6      31924540.7          291941       187180405  cudaMalloc
    6.1        82555753           6      13759292.2            2896        60919351  cudaDeviceSynchronize
    1.8        24567512           6       4094585.3           91088        22031229  cudaFree
    1.2        16613838           6       2768973.0           16708        16491051  cudaLaunchKernel
    0.0          349003           2        174501.5          174081          174922  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        82529902           2      41264951.0        21612253        60917649  conv_forward_kernel_2
    0.0            2816           2          1408.0            1344            1472  do_not_remove_this_kernel
    0.0            2752           2          1376.0            1280            1472  prefn_marker_kernel
```

▸ 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 [%]	78.19	Duration [msecond]	20.91
SOL Memory [%]	32.25	Elapsed Cycles [cycle]	25,085,152
SOL L1/TEX Cache [%]	32.25	SM Active Cycles [cycle]	25,082,382.61
SOL L2 Cache [%]	4.85	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	5.91	DRAM Frequency [cycle/usecond]	850.48

▸ 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 [%]	62.83	Duration [msecond]	60.38
SOL Memory [%]	28.15	Elapsed Cycles [cycle]	75,328,328
SOL L1/TEX Cache [%]	28.90	SM Active Cycles [cycle]	73,385,982.59
SOL L2 Cache [%]	2.26	SM Frequency [cycle/nsecond]	1.25
SOL DRAM [%]	2.42	DRAM Frequency [cycle/usecond]	884.30

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

CUDA Blogs and library