Name: Steven Y M Chang

NetID: sychang5

Section: AL2

ECE 408/CS483 Milestone 3 Report

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.185139 ms	0.66696 ms	0m0.220s	0.86
1000	1.70431 ms	6.35763 ms	0m0.316s	0.886
5000	8.44598 ms	31.8575 ms	0m0.816s	0.871

   Since the grading script is not compatible with streams and cannot correctly detect OP time (with stream implemented), I also included the layer times of the baseline milestone 2 implementation for ease of comparison with the stream optimization that I implement later in my report.

Batch Size	Layer Time 1	Layer Time 2	Total Layer Time
100	7.37142 ms	6.26589 ms	13.6373 ms
1000	62.2291 ms	50.5964 ms	112.826 ms
5000	309.108 ms	246.008 ms	555.116 ms

   The file used for milestone 2 baseline statistics is located in the *Project* folder within the *optimizations+baseline* folder, named *baseline.cu*.

2. **Optimization 1: FP16 Arithmetic (4 Points); File: *fp16_opt.cu* (Located in *Project* folder within *optimizations+baseline* folder)**

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

      I chose to implement the FP16 arithmetic optimization. I chose this optimization technique first because I thought that it would be the most straightforward technique that could be directly implemented in the baseline kernel by converting the values to half precision before performing the computation, then converting back to single precision float before storing as output. This could also be done without additional changes to the invocation or convolution setup, hence my first optimization. Moreover, this technique was also one of the most logical to me, as performing arithmetic operations on half precision values will be quicker than single precision float values due to reasons like reduced memory bandwidth requirements, with just a slight drop in accuracy.

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 implementation relies on using the half precision conversion and data movement library "cuda_fp16.h." Specifically, utilizing the __float2half(const float a) and __half2float(const __half a) functions to convert between half and float data type within the kernel function when performing computations (multiplication and accumulation). I converted the values retrieved from the input and mask into half precision before multiplying them and accumulating the result in a variable of half precision type. Then before storing the accumulated result into the output, I converted it back to single precision float.

I believe the optimization should theoretically increase performance of the forward convolution (decrease OP and execution times) as using half precision should increase the speed of the arithmetic computation compared to single precision float. Half precision computation should require less memory bandwidth and increased parallelism as values are only 16 bits compared to 32 bits.

This is my first optimization, so there were no other optimizations that I synergized/ merged with this technique. However, I do believe that this optimization should work alongside constant memory, TILE_WIDTH, and unrolling optimizations I attempt later on. This is because my implementation is directly within the kernel right before the computation and before storing the final result. Therefore, it should have no problem merging with the other optimizations even if the constant memory is of float type for instance. TILE_WIDTH also does not directly impact the data type and arithmetic operations, while unrolling should not change how many times we need to convert the values between the two data types.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.22766 ms	0.856756 ms	0m0.205s	0.86
1000	2.15007 ms	8.30696 ms	0m0.292s	0.887
5000	10.6309 ms	41.1852 ms	0m0.924s	0.8712

Statistics included above are based on FP16 optimization solely (built upon baseline implementation).

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).

No, implementing this optimization technique did not appear to improve overall performance. If we look at the OP and execution times, we can see that there was a general increase (worse performance) across various batch sizes. Specifically, the sum of OP times quite significantly increased (compared to the baseline) with this optimization implemented. Sum of OP times for batch size of 5000 increased by 11.5126 ms, increased by 2.39506 ms for batch size of 1000, and increased by 0.232321 ms for batch size of 100.

Kernel Statistics with FP16 Optimization Implemented (Batch Size of 5000)

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name

-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
----
  100.0        51823326           6       8637221.0            7872        41156518  conv_forward_kernel

    0.0            2816           2          1408.0            1376            1440  do_not_remove_this_kernel

    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel
```

Kernel Statistics for Baseline Implementation (Batch Size of 5000)

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name

-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
----
  100.0        40495964           6       6749327.3            7584        31955251  conv_forward_kernel

    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kernel

    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```

Since my optimization was directly implemented inside the forward convolution kernel, it makes sense to observe the performance results of the kernel to attempt to understand this increase in OP time and total execution time (worse performance). Specifically, I looked at the kernel statistics profiling results from *nsys* for batch size of 5000 (larger batch size for more distinct variations in results). As we can see from the results above, the average time of the forward convolution kernel is almost 2,000,000 ns greater with FP16 optimization implemented compared to the baseline implementation from milestone 2. In other words, by converting to half precision before computation, then converting back to single precision float before storing into the output within our kernel, we are actually slowing down the kernel.

Observing the kernel statistics helps us narrow the problem down to the actual conversion and computation. We know that the computation should be sped up with half precision compared to single precision, which means that the likely explanation for this decrease in performance is that the conversion between half precision and single precision took up significant time and was greater than the time saved by performing arithmetic operations using half precision. In other words, the conversion time outweighed the calculation speedup to yield an overall worse performance.

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

To implement this technique, I referenced the CUDA Math API Documentation, specifically on "Half Precision Conversion and Data Movement." Link to the documentation is as follows: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html#group__CUDA__MATH____HALF__MISC_1gd9d1945a39d5305f8e8a9c4bee0443e6. In addition, I also referenced Campuswire posts and responses by TAs/CAs regarding FP16 optimization (https://campuswire.com/c/G184FB646/feed/724).

3. **Optimization 2: Weight Matrix (Kernel Values) in Constant Memory (0.5 Points); File: *constant_mem_opt.cu* (Located in *Project* folder within *optimizations+baseline* folder)**

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

For my second optimization technique, I chose to implement the constant memory for storing the weight matrix. I chose this technique since it was very straightforward and could help me immediately analyze its impact on execution time in relation to the baseline of milestone 2 (did not implement with FP16 since my optimization approach for FP16 did not improve performance, hence starting over and stacking new optimizations to improve performance). It is also a simple technique that merely required me to declare constant memory for the weight matrix and copy the mask values to it before accessing it within the convolution 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?

The optimization works by storing the weight matrix in constant memory instead of allocating space on the device. Instead of copying the host_mask data to device using cudaMemcpy from Host to Device, we copy the data into constant memory using cudaMemcpyToSymbol, then we can access the mask values from within the kernel.

I believe that this optimization should increase performance of the forward convolution. This is because threads can read from constant memory (with caching) much faster than from global memory. By making the weight matrix constant (using __constant__), we tell the GPU that caching is safe (since the mask is read-only within the kernel and is not changed during execution). This alleviates the potential constraint posed by the global memory bandwidth on execution time when we access data from global memory, hence improving the performance overall through data access speedup.

While this optimization can technically synergize with the FP16 optimization that I first implemented, since we are simply storing the mask in constant memory instead of device memory, I decided to not build the constant memory optimization upon the FP16 optimization. This is because the FP16 optimization did not improve the performance (rather worsened it). On the other hand, this optimization can be merged with sweeping TILE_WIDTH and tuning with restrict and loop unrolling as will be implemented later on. This is because storing weight matrix in constant memory does not directly impact the kernel launch or kernel itself, where the other two implementations will mostly impact.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.170104 ms	0.651001 ms	0m0.278s	0.86
1000	1.55904 ms	6.36445 ms	0m0.308s	0.886
5000	7.71755 ms	31.7201 ms	0m0.856s	0.871

Statistics included above are based on Constant Memory optimization solely (built upon baseline implementation).

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).

Yes, implementing this optimization was successful in improving performance. While the total execution time did not vary by much, we can still observe a notable decrease in the sum of the OP times with constant memory optimization implemented. The sum of OP times decreased (improved) across all batch sizes, for batch size of 5000 decreased by 0.8658 ms, decreased by 0.13845 ms for batch size of 1000, and decreased by 0.030994 ms for batch size of 100.

Kernel Statistics with Constant Memory Optimization (Batch Size of 5000)

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        39303230           6       6550538.3            7328        31471435  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  do_not_remove_this_kernel
    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel
```

Kernel Statistics for Baseline Implementation (Batch Size of 5000)

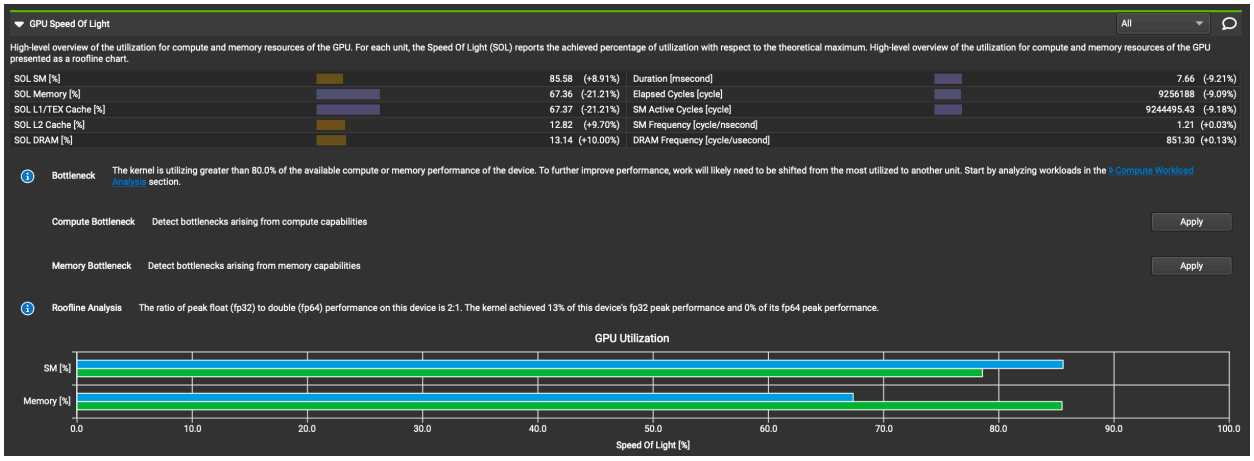
```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        40495964           6       6749327.3            7584        31955251  conv_forward_kernel
    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```

From the *nsys* profiling results (using batch size of 5000 for more distinct variations in results), we can see a notable decrease in average time of the forward convolution kernel, which is what we expected. In other words, we successfully improved the performance of the kernel itself, likely because of our optimization speeding up data access of mask values from constant memory during computation. For a better understanding of what is happening, we can reference the results from *Nsight-Compute* and compare it with the baseline as shown below.

Profiling Results with Optimization (Blue) from *Nsight-Compute* Compare to Baseline (Green)



From the results above (results with constant memory optimization shown in blue; results from baseline implementation shown in green), we note a few distinct statistics. The SM GPU Utilization percentage increased (from 78.58% to 85.58%), while the Memory GPU Utilization percentage decreased significantly (from 85.49% down to 67.36%). In addition, SOL L2 Cache percentage increased by nearly 10%, which may be the indication that we are efficiently accessing the weight matrix/mask with caching from constant memory. The significant reduction in memory utilization percentage can then be attributed to the efficient caching of the mask in constant memory that reduces the need for global memory transactions. Similarly, the SM utilization percentage increasing may be a result of the efficiency data access that reduces the wait time for memory transactions/transfers. These metrics, together, help explain the performance improvement of the convolution kernel, which is likely achieved, mainly, through more efficient data accesses through caching and constant memory.

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

To implement this technique, I referenced MP4 of the course to get a refresher on the use of constant memory, such as where to declare and initialize its size, how to copy data into it using cudaMemcpyToSymbol. In addition, I also referenced Lecture 7 on Convolution and Constant Memory to have a better understanding of the performance benefits of utilizing constant memory with caching as well as implementation tips.

4. **Optimization 3: Sweeping Parameters to Find Best Value (0.5 Points); File: *constant_mem_tilewidth_sweep_opt.cu* (Located in *Project* folder within *optimizations+baseline* folder)**

   NOTE: Sweeping Parameters Optimization, specifically, is located at the top of the file where TILE_WIDTH is defined.

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

      After implementing the constant memory for weight matrix optimization, I chose to implement the sweeping parameters optimization, specifically adjusting the block size (TILE_WIDTH) to attempt to obtain maximum performance. I chose this technique because it was one of the most straightforward techniques that I knew should function and allow me to build more optimizations on top of. It simply involved modifying the block size (TILE_WIDTH), which I defined altogether at the top of the file and observing changes in performance with varying sizes.

   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 sweeping/testing various values for parameters, specifically block size (TILE_WIDTH) for my optimization, then observing its impact on OP times and overall performance of the kernel. I do think that this optimization will increase performance of the forward convolution, if not at least impact the performance in some way, as adjusting the block size impacts kernel launch configuration and the way we access data, notably impacting memory access coalescing. Therefore, I believe that experimenting with different block sizes can positively improve performance.

      This optimization does in fact synergize with other optimizations. In addition to building this optimization upon the constant memory optimization described above, I also go on to build the tuning with restrict and loop unrolling optimization on top of this technique. This optimization synergizes with the other two as the optimizations involve modifications to different aspects of the code. While constant memory modifies from where we access the data, and block size impacts kernel launch and how the threads access the data, unrolling simply involves writing out the arithmetic operations sequentially instead of nesting them in loops. Hence, I believe that these optimizations should work together.

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

For this optimization, I tested several values for TILE_WIDTH, namely 18, 19, and 20 to find the optimal performance compared to the baseline implementation with TILE_WIDTH of 16. The performance for the various parameter values is noted below.

TILE_WIDTH of 18

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.211884 ms	0.498903 ms	0m0.299s	0.86
1000	1.95002 ms	4.58852 ms	0m0.276s	0.886
5000	9.69125 ms	22.9091 ms	0m1.020s	0.871

TILE_WIDTH of 19

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.219952 ms	0.499942 ms	0m0.168s	0.86
1000	1.85829 ms	4.3612 ms	0m0.292s	0.886
5000	10.1289 ms	23.8662 ms	0m0.912s	0.871

TILE_WIDTH of 20

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.181784 ms	0.51999 ms	0m0.236s	0.86
1000	1.66737 ms	4.95059 ms	0m0.312s	0.886
5000	8.26475 ms	24.7054 ms	0m0.888s	0.871

Statistics included above are based on Sweeping and Constant Memory optimization (built upon Optimization 2: Weight Matrix (Kernel Values) in Constant Memory).

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).

Yes, this optimization successfully improved performance, when comparing with the previous constant memory optimization on which this sweeping optimization was built on. I swept and tested various values for the block size (TILE_WIDTH) as shown above. We can note from the results that while the OP time for layer 1 did not change significantly, the OP time for layer 2 decreased quite significantly, approximately 8 ms for batch sizes of 5000 (roughly 20% decrease), 1.5 ms for batch sizes of 1000 (roughly 18% decrease), and 0.1 ms for batch size of 100 (roughly 12% decrease). After sweeping various values for TILE_WIDTH as shown above, while the OP times improved by approximately the same amount (across different block sizes) from the baseline from optimization 2, the total execution time for batch size of 5000 was notably lower for TILE_WIDTH of 20. Hence, I chose TILE_WIDTH of 20 as the optimal value for closer comparison with the previous optimization.

Kernel Statistics with Sweeping TILE_WIDTH = 20 Optimization (Batch Size of 5000)

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        33241205           6       5540200.8            7488        24836952  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2560           2          1280.0            1216            1344  prefn_marker_kernel
```

Kernel Statistics with Constant Memory Optimization (Batch Size of 5000)

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        39303230           6       6550538.3            7328        31471435  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  do_not_remove_this_kernel
    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel
```

Observing the *nsys* profiling results above, we can directly compare the forward convolution kernel times between the baseline presented by optimization 2 (constant memory optimization), with our newly implemented optimization sweeping block size (TILE_WIDTH), specifically for the optimal value of 20 that we decided above based on execution times. We take note of a significant decrease in the average time of the kernel by approximately 1,000,000 ns, which represents around a 15% decrease. Hence, the decrease in OP times can be attributed to an overall decrease in the convolution kernel time, which may be due to reasons aforementioned regarding more efficient memory coalescing impacted by adjusting block size.

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

Since this optimization was very straightforward, I did not reference particular sources when implementing this technique aside from a Campuswire post clarifying the requirements (https://campuswire.com/c/G184FB646/feed/756).

5. **Optimization 4: Tuning with Restrict and Loop Unrolling (3 Points); File: *constant_mem_tilewidth_sweep_unroll_opt.cu* (Located in *Project* folder within *optimizations+baseline* folder)**

   FINAL SUBMITTED VERSION (ALSO IN *new-forward.cu*)
   NOTE: Tuning with Restrict and Loop Unrolling Optimization specifically is located at the function header of conv_forward_kernel (applied to the input, output, and mask pointer arguments), as well as in the kernel function itself within the for loop that loops through the channels.

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

      With constant memory for weight matrix/mask and sweeping parameters (block size/ TILE_WIDTH) optimizations implemented, I chose to implement the tuning with restrict and loop unrolling technique. I chose to implement this technique because I believed that it was one that would synergize well with my previous implementations, since it doesn't impact kernel launch and the modification is within the kernel itself, which I have not changed significantly yet.

   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?

      There are two main aspects to this optimization: restrict and unrolling. With the former, we make the pointers to the input, output, and mask to be restricted to tell the compiler that the pointers are not aliased (essentially not overlapping in memory). This can lead to reduced memory accessed and reduced computation times by avoiding redundant loads from memory and allowing the compiler to reorder and perform common operations to optimize performance, which should be particularly useful and improve performance given our significant number of arithmetic operations. On the other hand, I believe that unrolling should also theoretically help optimize performance as it is easier for the compiler to execute sequential code than for loops. In other words, to implement unrolling, I got rid of the for loops for p and q and manually wrote out the arithmetic operations, unrolling for K less than a certain optimal value, with that optimal value determined through multiple tests as will be included below. For all the reasons above, I do believe that this optimization will increase performance of the forward convolution.

      In addition, this optimization does synergize with previous optimizations, namely the constant memory for weight matrix and sweeping various parameters (TILE_WIDTH) optimizations. This is because the constant memory optimization simply changes where we access the data from, the TILE_WIDTH mainly impacts kernel launch configurations and accessing of data, while unrolling is essentially just writing out each arithmetic operation instead of nesting it inside loops, which bars no particular impact on the other optimizations mentioned.

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

For this optimization, tuning is a key aspect, as such I tested various upper bound values for K which we unroll (unroll if K is less than or equal to a certain number) in order to find the optimal performance. The performance for various values when unrolling and its results are noted below.

Unroll if K <= 3

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.19021 ms	0.539124 ms	0m0.180s	0.86
1000	1.73126 ms	5.12493 ms	0m0.317s	0.886
5000	8.59 ms	25.5753 ms	0m0.860s	0.871

Unroll if K <= 8

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.124465 ms	0.385586 ms	0m0.213s	0.86
1000	1.09403 ms	3.57825 ms	0m0.344s	0.886
5000	5.37434 ms	17.8049 ms	0m0.904s	0.871

Unroll if K <= 10

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.124429 ms	0.380211 ms	0m0.212s	0.86
1000	1.09563 ms	3.5635 ms	0m0.506s	0.886
5000	5.39017 ms	17.6637 ms	0m0.920s	0.871

Unroll if K <= 11

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.124233 ms	0.37442 ms	0m0.144s	0.86
1000	1.17393 ms	3.68958 ms	0m0.280s	0.886
5000	5.81294 ms	18.3875 ms	0m1.020s	0.871

Statistics included above are based on Tuning with Restrict and Unrolling, Sweeping, and Constant Memory optimization (built upon Optimization 3: Sweeping Parameters to Find Best Value).

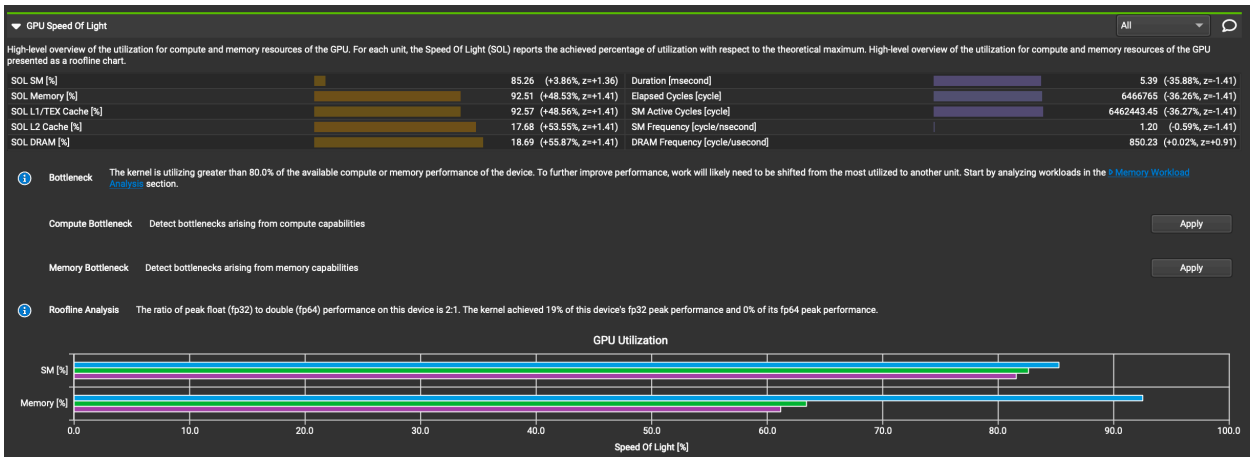
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, implementing this optimization was very successful in improving performance. In particular, we can observe the sum of the OP times and compare the results with the previous optimization (Optimization 3: Sweeping Parameters to Find the Best Value), specifically results from the optimal value of TILE_WIDTH = 20 that we chose to build this optimization on.

In the process of tuning, I first unrolled if K was less than or equal to 3. The result was that there was actually an increase in the sum of OP times across the 3 batch sizes. As such, I tuned it to a higher value and experimented at several values like 8, 10, and 11. With these values, the sum of the OP times decreased significantly, by nearly 10 ms for batch size of 5000, by approximately 2 ms for batch size of 1000, and by approximately 0.2 ms for batch size of 100.

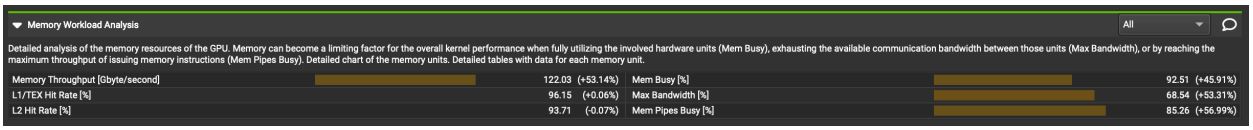
To better understand where the improvement is stemming from, we can look at the *Nsight-Compute* profiling results, specifically comparing the results from the previous optimization (without restrict and tuning loop unroll), and the results from unrolling if K <= 3, and unrolling if K <= 10 (which I chose as the optimal value based on the OP times results observed above).

Profiling Results Unrolling if K <= 10 (Blue), Unrolling if K <= 3 (Purple) from *Nsight-Compute* Compare to Baseline of Previous Optimization (Green)

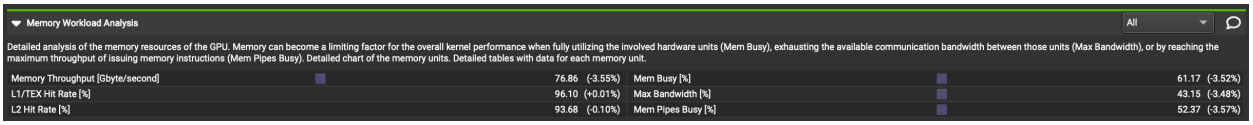


As can be seen above, the biggest distinction between the three results (particularly with optimization K <= 10 that is highlighted in blue in GPU utilization and represented by the statistics shown at the top), is the memory utilization percentage. This may be due to the increase use of registers from the unrolling during execution. Paying specific attention to memory, we further look at the memory throughput, which was significantly different with the optimization in place (for the optimal value of unrolling if K <= 10), as shown below

Memory Workload Unrolling if K <= 10 Compared to Baseline of Previous Optimization



Memory Workload Unrolling if K <= 3 Compared to Baseline of Previous Optimization



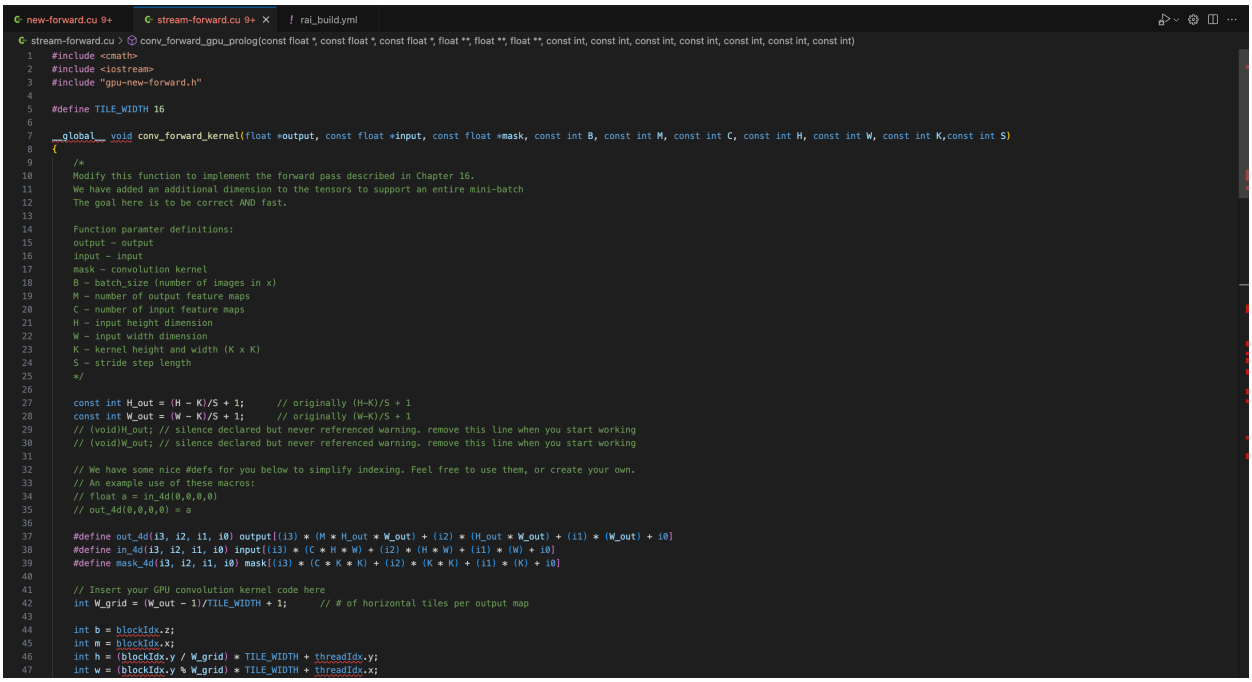
The results above demonstrate that there was a significant memory throughput increase (more than 50% compared to the previous optimization) when we unrolled for K <= 10, while there was not significant change (rather slight decrease) for when we unrolled for K <= 3. This increase in memory throughput may have to do with more efficient memory access patterns and better caching outcomes, which is also reflected in the GPU Speed of Light statistics above, thereby leading to a drastic increase in performance when we tuned the loop unrolling to the optimal value of 10 (unroll if K <= 10).

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

On the restricting aspect of the optimization, I referenced a Campuswire post (https://campuswire.com/c/G184FB646/feed/746), which pointed me to the CUDA Programming Guide on __restrict__ (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrict) that detailed usage and motivation for restricting pointers. In addition, I also referenced external sources on loop unrolling, regarding its implementation and benefits (https://www.sciencedirect.com/topics/computer-science/loop-unrolling#:~:text=Loop%20unrolling%20).

6. **Optimization 5: Streams to Overlap Computation with Data Transfer (4 Points); File: *stream-forward.cu* (Located in *Project* folder within *optimizations+baseline* folder)**

Code for Streams:



```
#include <cmath>
#include <iostream>
#include "gpu-new-forward.h"

#define TILE_WIDTH 16

__global__ void conv_forward_kernel(float *output, const float *input, const float *mask, const int B, const int M, const int C, const int H, const int W, const int K,const int S)
{
    /*
    Modify this function to implement the forward pass described in Chapter 16.
    We have added an additional dimension to the tensors to support an entire mini-batch
    The goal here is to be correct AND fast.

    Function paramter definitions:
    output - output
    input - input
    mask - convolution kernel
    B - batch_size (number of images in x)
    M - number of output feature maps
    C - number of input feature maps
    H - input height dimension
    W - input width dimension
    K - kernel height and width (K x K)
    S - stride step length
    */

    const int H_out = (H - K)/S + 1;      // originally (H-K)/S + 1
    const int W_out = (W - K)/S + 1;      // originally (W-K)/S + 1
    // (void)H_out; // silence declared but never referenced warning. remove this line when you start working
    // (void)W_out; // silence declared but never referenced warning. remove this line when you start working

    // We have some nice #defs for you below to simplify indexing. Feel free to use them, or create your own.
    // An example use of these macros:
    // float a = in_4d(0,0,0,0)
    // out_4d(0,0,0,0) = a

    #define out_4d(i3, i2, i1, i0) output[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
    #define in_4d(i3, i2, i1, i0) input[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here
    int W_grid = (W_out - 1)/TILE_WIDTH + 1;      // # of horizontal tiles per output map

    int b = blockIdx.z;
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
```

```
    float acc = 0.0f;
    if(h < H_out && w < W_out){
        for(int c = 0; c < C; c++){
            for(int p = 0; p < K; p++){
                for(int q = 0; q < K; q++){
                    acc += in_4d(b, c, S*h+p, S*w+q) * mask_4d(m, c, p, q);
                }
            }
        }
    out_4d(b, m, h, w) = acc;
    }

    #undef out_4d
    #undef in_4d
    #undef mask_4d
}
```

```
__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input, const float *host_mask, float **device_output_ptr, float **device_input_ptr, float **device_mask_ptr, const int B, const
{
    // Allocate memory and copy over the relevant data structures to the GPU

    // We pass double pointers for you to initialize the relevant device pointers,
    //  which are passed to the other two functions.

    // Useful snippet for error checking
    // cudaError_t error = cudaGetLastError();
    // if(error != cudaSuccess)
    // {
    //     std::cout<<"CUDA error: "<<cudaGetErrorString(error)<<std::endl;
    //     exit(-1);
    // }

    // find output width and height of each feature
    const int H_out = (H - K)/S + 1;
    const int W_out = (W - K)/S + 1;

    // find size of input, output, and mask
    int size_of_input = B*C*H*W*sizeof(float);
    int size_of_output = B*M*H_out*W_out*sizeof(float);
    int size_of_mask = M*C*K*K*sizeof(float);

    // allocate device memory for input, output, and mask
    cudaMalloc((void**) device_input_ptr, size_of_input);
    cudaMalloc((void**) device_output_ptr, size_of_output);
    cudaMalloc((void**) device_mask_ptr, size_of_mask);




    /* Stream Optimization */
    // Per Campuswire TA/CA recommendation, set number of streams to be batch size B
```

```
    // pin host input and output memory; use size in bytes of input and output and default flag mode
    cudaHostRegister((void*) host_input, size_of_input, cudaHostRegisterDefault);
    cudaHostRegister((void*) host_output, size_of_output, cudaHostRegisterDefault);

    // Set the kernel dimensions
    float H_grid = ceil((float)H_out/TILE_WIDTH);      // # of vertical tiles per output map
    float W_grid = ceil((float)W_out/TILE_WIDTH);      // # of horizontal tiles per output map
    dim3 dimGrid(M, H_grid*W_grid, B/B);               // need to adjust grid dimension to account for segment size (divide by B (number of streams) for size of each segment)
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

    // Declare and create streams (B streams per Campuswire recommendation)
    cudaStream_t stream[B];
    for(int i = 0; i < B; i++){
        cudaStreamCreate(&stream[i]);
    }
    cudaDeviceSynchronize();

    // find segment offset for pointer
    int input_segment = (B*C*H*W)/B;
    int output_segment = (B*M*H_out*W_out)/B;

    // Transfer Data and Launch Kernel; Loop through each stream/segment
    for(int i = 0; i < B; i++){
        // copy input and mask data to device memory
        cudaMemcpyAsync(*device_mask_ptr, host_mask, size_of_mask, cudaMemcpyHostToDevice, stream[i]);
        // need to begin to transfer data from offset into the input array/matrix, which is given by the current stream segment (multiply by size of each segment)
        cudaMemcpyAsync(*device_input_ptr + i*input_segment, host_input + i*input_segment, input_segment*sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        // launch kernel with current stream (offset into device output and input by current stream segment)
        conv_forward_kernel<<<dimGrid, dimBlock, 0, stream[i]>>>(*device_output_ptr + i*output_segment, *device_input_ptr + i*input_segment, *device_mask_ptr, B, M, C, H, W, K, S);
        // copy output back to host memory
        cudaMemcpyAsync((float*) host_output + i*output_segment, *device_output_ptr + i*output_segment, output_segment*sizeof(float), cudaMemcpyDeviceToHost, stream[i]);
    }
    cudaDeviceSynchronize();

    // Destroy the streams
    for(int i = 0; i < B; i++){
        cudaStreamDestroy(stream[i]);
    }

    // unpin host input and output memory
    cudaHostUnregister((void*) host_input);
    cudaHostUnregister((void*) host_output);
}
```

```
__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input, const float *device_mask, const int B, const int M, const int C, const int H, const int W, const int K, const int S)
{
    // // Set the kernel dimensions and call the kernel
    // const int H_out = (H - K)/S + 1;
    // const int W_out = (W - K)/S + 1;
    // float H_grid = ceil((float)H_out/TILE_WIDTH);      // # of vertical tiles per output map
    // float W_grid = ceil((float)W_out/TILE_WIDTH);      // # of horizontal tiles per output map
    // // int H_grid = ((H_out -1)/TILE_WIDTH) + 1;      // # of vertical tiles per output map
    // // int W_grid = ((W_out -1)/TILE_WIDTH) + 1;      // # of horizontal tiles per output map
    // dim3 dimGrid(M, H_grid*W_grid, B);
    // dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

    // conv_forward_kernel<<<dimGrid, dimBlock>>>(device_output, device_input, device_mask, B, M, C, H, W, K, S);
}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_mask, const int B, const int M, const int C, const int H, const int W, const int K, const int
{
    // // Copy the output back to host
    // const int H_out = (H - K)/S + 1;
    // const int W_out = (W - K)/S + 1;
    // int size_of_output = B*M*H_out*W_out*sizeof(float);
    // cudaMemcpy(host_output, device_output, size_of_output, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(device_input);
    cudaFree(device_output);
    cudaFree(device_mask);
}


__host__ void GPUInterface::get_device_properties()
{
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    for(int dev = 0; dev < deviceCount; dev++)
    {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

        std::cout<<"Device "<<dev<<" name: "<<deviceProp.name<<std::endl;
        std::cout<<"Computational capabilities: "<<deviceProp.major<<"."<<deviceProp.minor<<std::endl;
        std::cout<<"Max Global memory size: "<<deviceProp.totalGlobalMem<<std::endl;
        std::cout<<"Max Constant memory size: "<<deviceProp.totalConstMem<<std::endl;
        std::cout<<"Max Shared memory size per block: "<<deviceProp.sharedMemPerBlock<<std::endl;
        std::cout<<"Max threads per block: "<<deviceProp.maxThreadsPerBlock<<std::endl;
        std::cout<<"Max block dimensions: "<<deviceProp.maxThreadsDim[0]<<" x, "<<deviceProp.maxThreadsDim[1]<<" y, "<<deviceProp.maxThreadsDim[2]<<" z"<<std::endl;
        std::cout<<"Max grid dimensions: "<<deviceProp.maxGridSize[0]<<" x, "<<deviceProp.maxGridSize[1]<<" y, "<<deviceProp.maxGridSize[2]<<" z"<<std::endl;
        std::cout<<"Warp Size: "<<deviceProp.warpSize<<std::endl;
    }
}
```

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

   For my fifth optimization, I chose to implement streams to overlap computation with data transfer. I chose this technique to attempt because it was thoroughly discussed during lecture and is a technique that should lead to improved performance if we have large input size and significant amount of computation in 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 making the computation within the kernel and data transfer between host and device memory overlap. In other words, rather than simply parallelizing the computation in the kernel, we also want to parallelize the entire data transfer and computation process. Instead of waiting for the input data and mask data to transfer to device memory, then performing the computation in the kernel, then transfer the result back to the host (sequential process), we want to transfer data in segments to the device, perform the computation, and transfer the data back to host, all at the same time (simultaneously execute the kernel while copying data to and from the device memory). I believe this optimization will increase performance of the forward convolution because we are parallelizing the originally sequential process of data transfer and computation.

In addition, per Campuswire post (https://campuswire.com/c/G184FB646/feed/667) that directed to CUDA runtime documentation (https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior), I also chose to pin the input and output host memory first before implementing memory copies and kernel launches. This is to prevent synchronization of the memory transfers, which would essentially take away the benefits of asynchronous memory transfer operations that the stream optimization is looking to utilize.

Referencing Campuswire post (https://campuswire.com/c/G184FB646/feed/679), I implemented the streaming optimization on its own (from the baseline implementation of milestone 2), since the grading script is not compatible with streaming and cannot correctly detect OP time. However, streams should theoretically synergize with other optimizations such as adjusting TILE_WIDTH and unrolling, since those modifications do not impact the division of kernel launches and memcpy into segments based on the number of streams we use.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time (sys)	Accuracy
100	0.001694 ms	0.001585 ms	0m0.196s	0.86
1000	0.005396 ms	0.006072 ms	0m0.308s	0.886
5000	0.006853 ms	0.008163 ms	0m0.916s	0.871

Since the grading script is not compatible with streaming and cannot correctly detect OP time, I also included the layer times for comparison with the baseline of milestone 2.

Batch Size	Layer Time 1	Layer Time 2	Total Layer Time
100	6.24476 ms	5.0214 ms	11.2662 ms
1000	53.9294 ms	35.3135 ms	89.2429 ms
5000	284.358 ms	181.956 ms	466.314 ms

Statistics included above are based on Stream optimization solely (built upon baseline implementation).

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).

Yes, it appears that implementing the stream optimization was successful in improving performance. While the OP times are inaccurate for the purpose of evaluating the success (or failure) or this optimization technique, we can look at the sum of the layer times compared to the baseline implementation and note that there was a decrease (performance improvement) across all 3 batch sizes. For batch size of 5000, total layer time decreased by almost 90 ms, for batch size of 1000, decreased by approximately 20 ms, and for batch size of 100, decreased by approximately 2 ms, which is between a 15% to 20% decrease between the different batch sizes.

If we recall that layer time includes the time for all kernel and CUDA API calls, this means that this decrease in layer time can likely be attributed to the increased performance of memory transfers overlapping with kernel executions that is achieved through stream optimization. Since stream optimization involves splitting kernel launches with smaller sizes, I also include results from *nsys* profiling below compared to that of the baseline of milestone 2 to illustrate the impacts of stream optimization.

CUDA API and Kernel Statistics from Stream Optimization

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   62.7      1222591768          20      61129588.4            2662      1219020563  cudaFree
   11.2       219207406          12      18267283.8          159343       114673799  cudaHostRegister
    9.3       181238638          20       9061931.9            2907       174335508  cudaMalloc
    5.9       115105148       10010         11499.0            1238         2772074  cudaStreamCreate
    5.2       101064074       30030          3365.4            2250         4357875  cudaMemcpyAsync
    2.3        44846879       10014          4478.4            3333         3329914  cudaLaunchKernel
    1.8        35349709          12       2945809.1           88697        14998291  cudaHostUnregister
    1.3        24647361       10010          2462.3            1493          382657  cudaStreamDestroy
    0.3         6220631          22        282756.0            1544         3969755  cudaDeviceSynchronize
    0.0          781260           2        390630.0           31121          750139  cudaMemcpy



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0       154392309       10010         15423.8            5952           37344  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  prefn_marker_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
```

CUDA API and Kernel Statistics from Baseline Implementation (Milestone 2)

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   65.7       523354678          20      26167733.9           30499       272803945  cudaMemcpy
   26.7       212978525          20      10648926.2            2688       208743692  cudaMalloc
    5.1        40523460          10       4052346.0            3589        31957561  cudaDeviceSynchronize
    1.9        15084836          10       1508483.6           23982        14818197  cudaLaunchKernel
    0.6         4760702          20        238035.1            3746         2119856  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        40495964           6       6749327.3            7584        31955251  conv_forward_kernel
    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```

Specifically, we should take note of the number of calls on the asynchronous memory copy as well as the number of instances of the forward convolution kernel and the significant drop in average time. The significant increase in these numbers in the optimized results indicate that we are indeed breaking up the data transfers and kernel launches into smaller 'batches,' which would help facilitate overlapping or simultaneous execution between the two (transfer of data and kernel execution) that leads to a performance increase and speedup.

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

To implement this technique, I referenced Lecture 22 on GPU Data Transfer, focusing in particular on declaration and usage of cudaStream_t, cudaStreamCreate, and cudaMemcpyAsync. In addition, I also referenced Campuswire posts about pinning host input and output memory for performance improvement (https://campuswire.com/c/G184FB646/feed/667), and setting number of streams to batch size (https://campuswire.com/c/G184FB646/feed/744 and https://campuswire.com/c/G184FB646/feed/763). A blog on CUDA streams also helped me brainstorm implementation approaches (https://leimao.github.io/blog/CUDA-Stream/). Moreover, I referenced CUDA memory management API documentation linked below: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8d5c17670f16ac4fc8fcb4181cb490c for information regarding pinning memory.