Forward Pass Project Results

0. Batch profiling results for basic forward convolution kernel:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*3.90226 ms*	*5.1347 ms*	*1.276 s*	*.86*
1000	*11.0889 ms*	*48.1185 ms*	*11.516 s*	*.886*
10000	*23.2205 ms*	*146.022 ms*	*1 min 40.687 s*	*.8714*

1. ***Weight matrix in constant memory***

    a. Which optimization did you choose to implement? Explain why did you choose that optimization technique.
    I chose to implement a weight matrix in constant memory because I thought it would make the kernel more memory efficient since all of the operations can be stored and reused when needed from one single location.

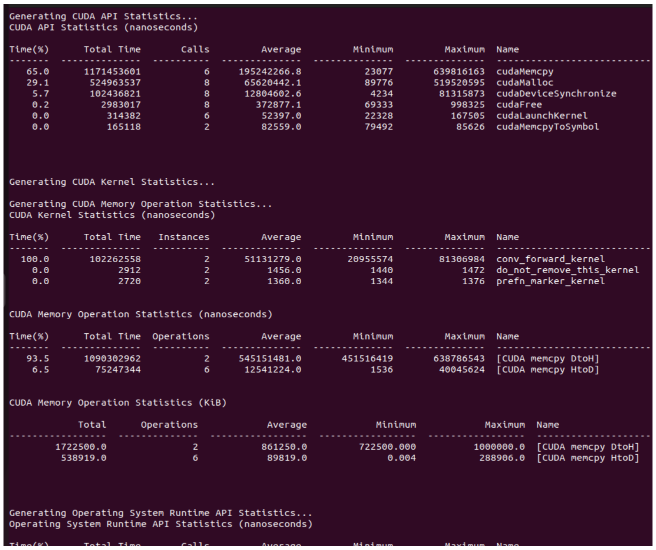
    b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?
    This optimization works by utilizing a matrix placed in constant memory to act as a filter over the various input images in the fashion data set. It is very memory efficient as constant memory is faster to access thus reducing the bandwidth requirements. I thought that this optimization would increase performance of the forward convolution since the data we would be dealing with would be anywhere from 100 to 10000 in terms of batch size, so the convolution would be expedited if there were less memory transactions required due to the matrix allowing better data accessibility. This optimization synergizes well with tiling and other shared memory techniques.

    c. Batch profiling results:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*.226164 ms*	*.839785 ms*	*0m 1.220s*	*.86*
1000	*2.10433 ms*	*8.13764 ms*	*0m 10.177s*	*.886*
10000	*20.6525 ms*	*81.62 ms*	*1m 40.276s*	*.8714*

d. Was implementing this optimization successful in improving performance? Why or why not?

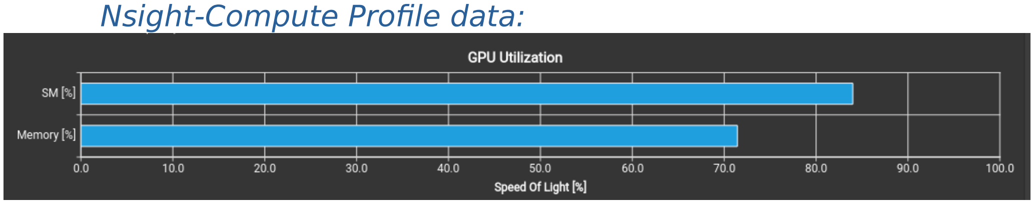
nsys Profile data:

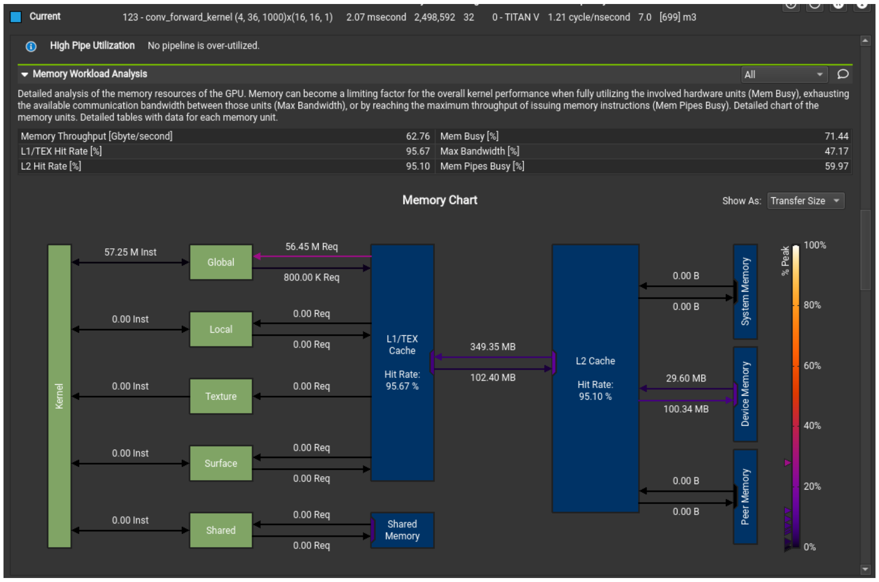


```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------   --------------  ----------  --------------  --------------  --------------  --------------------------
   65.0       1171453601           6     195242266.8           23077       639816163  cudaMemcpy
   29.1        524963537           8      65620442.1           89776       519520595  cudaMalloc
    5.7        102436821           8      12804602.6            4234        81315873  cudaDeviceSynchronize
    0.2          2983017           8        372877.1           69333          998325  cudaFree
    0.0           314382           6         52397.0           22328          167505  cudaLaunchKernel
    0.0           165118           2         82559.0           79492           85626  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------   --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        102262558           2      51131279.0        20955574        81306984  conv_forward_kernel
    0.0             2912           2          1456.0            1440            1472  do_not_remove_this_kernel
    0.0             2720           2          1360.0            1344            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------   --------------  ----------  --------------  --------------  --------------  --------------------------
   93.5       1090302962           2     545151481.0       451516419       638786543  [CUDA memcpy DtoH]
    6.5         75247344           6      12541224.0            1536        40045624  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average             Minimum             Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)
```

The time (%) displayed by the CUDA Kernel Statistics show that 65.0% of time was taken by the cudaMemcpy. Compared to the figure of 71.6% of time needed for cudaMemcpy from the baseline code measured for Milestone 2, this means that there was a reduction of over 6% in time. This can be attributed to successfully implementing the weighted matrix in constant memory which is why cudaMalloc increases from 15.7% to 29.1%. We can conclude that this optimization was successful due to lowered OP times (~102ms) and the reduction in time taken by cudaMemcpy.

Nsight-Compute Profile data:





Based on the Nsight-Compute data, we can observe that there were high amounts of SM and Memory utilization at figures of approximately 85% and 70% respectively which are relatively high figures compared to 75% and 60% respectively from Milestone 2, but it is important to look at the memory workload analysis. Judging by how the L1 and L2 Cache have hit rates of approximately 95%, this means that the optimization of implementing a weighted matrix in constant memory is memory efficient and we were successful in increasing performance.

e. Modified code:

```
#define TILE_WIDTH 16

__constant__ float matrix[10000];
__global__ void conv_forward_kernel(float *output, const float *input, const float *mask, const int Batch, const int Map_out, const int Channel, con
{
    /*
    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
    Batch - batch_size (number of images in x)
    Map_out - number of output feature maps
    Channel - number of input feature maps
    Height - input height dimension
    Width - input width dimension
    K - kernel height and width (K x K)
    */

    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int W_grid = (Width_out+TILE_WIDTH-1)/TILE_WIDTH;

    // 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) * (Map_out * Height_out * Width_out) + (i2) * (Height_out * Width_out) + (i1) * (Width_out) + i0]
    #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * Width) + (i1) * (Width) + i0]
    #define mask_4d(i3, i2, i1, i0)  matrix[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
    int b = blockIdx.z;

    float acc = 0.0f;
    for(int c = 0; c < Channel; c++) { // sum over all input channels
        for(int p = 0; p < K; p++) {// loop over KxK filter
            for(int q = 0; q < K; q++)
                acc += in_4d(b, c, h + p, w + q) * mask_4d(m, c, p, q);
```

Code also modified in conv_forward_gpu_prolog.

```
__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input, const floa
**device_mask_ptr, const int Batch, const int Map_out, const int Channel, const int Height, const int Width, cons
{
    // 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);
    // }

    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    cudaMalloc((void **) device_output_ptr, Batch*Map_out*Height_out*Width_out*sizeof(float));
    cudaMalloc((void **) device_input_ptr, Batch*Channel*Height*Width*sizeof(float));
    cudaMalloc((void **) device_mask_ptr, Map_out*Channel*K*K*sizeof(float));
    cudaMemcpy(*device_input_ptr, host_input, Batch*Channel*Height*Width*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(matrix, host_mask, Map_out*Channel*K*K*sizeof(float));
}
```

2. *Tiled shared memory convolution*

a. Which optimization did you choose to implement? Explain why did you choose that optimization technique.
The next optimization I chose to implement was a tiled shared memory convolution because I thought it would allow for more hardware resources like memory to be used efficiently and enable parallelism across multiple threads at the same time.

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 dividing data into rectangular tiles and storing them in shared memory where convolution is carried out on all tiles simultaneously. I thought this optimization would increase performance of the forward convolution since it would reduce the bottleneck of memory bandwidth requirements by improving cache utilization and take advantage of parallelism. This optimization can synergize well with the implementation of a weighted matrix in constant memory as both techniques have a common goal of increasing memory efficiency and reducing data fetching.

c. Batch profiling results:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*.312647 ms*	*.675611 ms*	*0m 1.263s*	*.86*
1000	*2.97889 ms*	*6.45686 ms*	*0m 9.802s*	*.886*
10000	*29.624*	*64.7542*	*1m*	*.8714*

	ms	*ms*	*36.691s*	

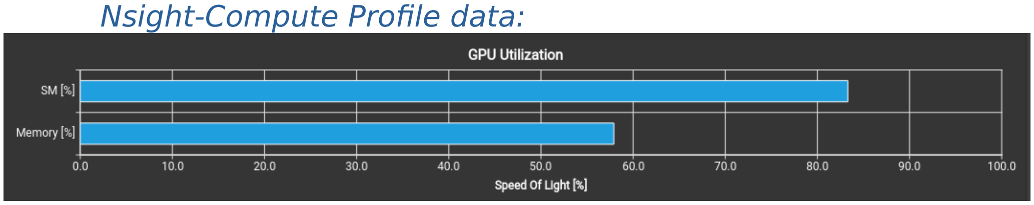
d. Was implementing this optimization successful in improving performance? Why or why not?

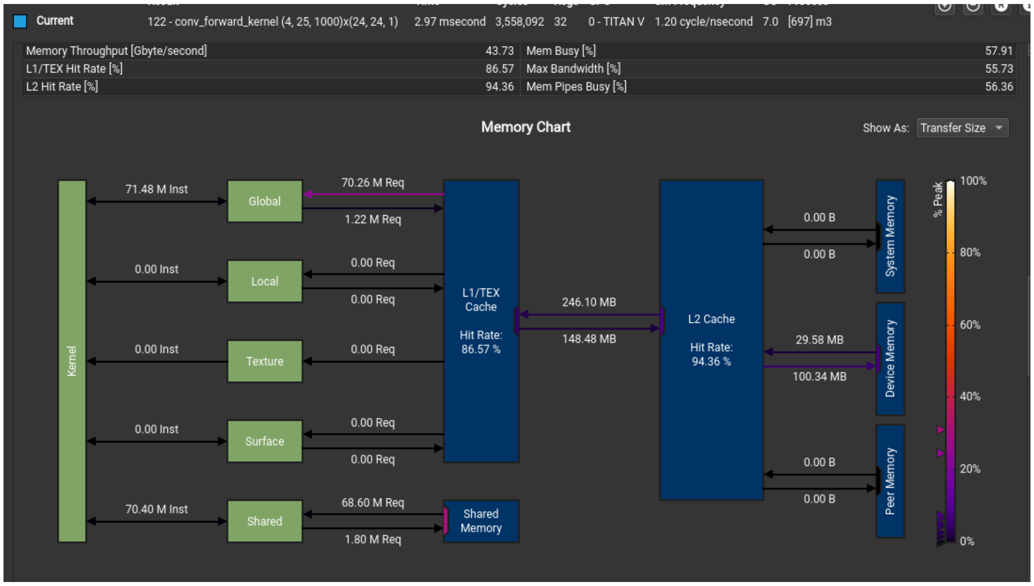
nsys Profile data:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
   77.6      1049829433           8     131228679.1           17860       564702556  cudaMemcpy
   13.9       188041322           8      23505165.2           68288       183451017  cudaMalloc
    7.0        95136600           6      15856100.0            3070        65019306  cudaDeviceSynchronize
    1.2        16406189           6       2734364.8           16502        16289085  cudaLaunchKernel
    0.2         2679011           8        334876.4           63913          865837  cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
  100.0        94997359           2      47498679.5        29980691        65016668  conv_forward_kernel
    0.0            2720           2          1360.0            1344            1376  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
   91.4       955420106           2     477710053.0       391523807       563896299  [CUDA memcpy DtoH]
    8.6        89529435           6      14921572.5            1216        47993284  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  -------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
```

Based on the data displayed for the CUDA API Statistics, cudaMemcpy takes 77.6% time which is approximately 5% more usage of that API call from Milestone 2 at 71.6% and additionally the cudaDeviceSynchronize went down from 11.2% to 7.0%. This data could be attbuted to utilizing more shared memory and the over 4% reduction in the cudaDeviceSynchronize could be interpreted as the code parallelizing more, thus the kernel having less of a need for that API call and the optimization being successful as also justified by the reduced OP time of approximately 95 ms.

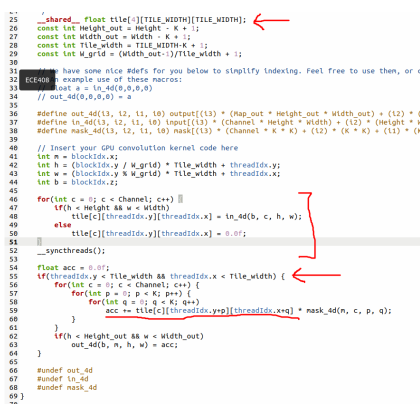
Nsight-Compute Profile data:





This Nsight-Compute data shows that this optimization utilizes the same amount of SM as the weighted matrix in constant memory implementation (thus about approximately 7% more SM than Milestone 2) but actually uses less Memory at approximately 58% versus ~60% utilization for Milestone 2. This means that the tiled implementation in shared memory is a boost in performance as it makes better use of the memory bandwidth and requires less accesses.

e. Modified code:



```
    __shared__ float tile[4][TILE_WIDTH][TILE_WIDTH];
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int Tile_width = TILE_WIDTH-K + 1;
    const int W_grid = (Width_out-1)/Tile_width + 1;

    // We have some nice #defs for you below to simplify indexing. Feel free to use them, or c
    // 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) * (Map_out * Height_out * Width_out) + (i2) * (
    #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * W
    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K

    // Insert your GPU convolution kernel code here
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * Tile_width + threadIdx.y;
    int w = (blockIdx.y % W_grid) * Tile_width + threadIdx.x;
    int b = blockIdx.z;

    for(int c = 0; c < Channel; c++) {
        if(h < Height && w < Width)
            tile[c][threadIdx.y][threadIdx.x] = in_4d(b, c, h, w);
        else
            tile[c][threadIdx.y][threadIdx.x] = 0.0f;
    }
    __syncthreads();

    float acc = 0.0f;
    if(threadIdx.y < Tile_width && threadIdx.x < Tile_width) {
        for(int c = 0; c < Channel; c++) {
            for(int p = 0; p < K; p++) {
                for(int q = 0; q < K; q++)
                    acc += tile[c][threadIdx.y+p][threadIdx.x+q] * mask_4d(m, c, p, q);
            }
        }
        if(h < Height_out && w < Width_out)
            out_4d(b, m, h, w) = acc;
    }

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

3. Tuning with restrict and loop unrolling

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

I chose this optimization technique because by restricting pointers and unrolling the main for loop in the kernel I thought that performance would be increased due to the reduction of memory access latency.

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 allowing the compiler to optimize memory accesses by assuming that memory pointed to by a restricted pointer is not aliased by any other pointer. This means this optimization would increase the performance of forward convolution since the there would be improved cache utilization and reduced memory access latency. Loop unrolling would also boost performance as it would increase task parallelism. Another type of optimization that could go along with this technique is kernel fusion.

c. Batch profiling results:

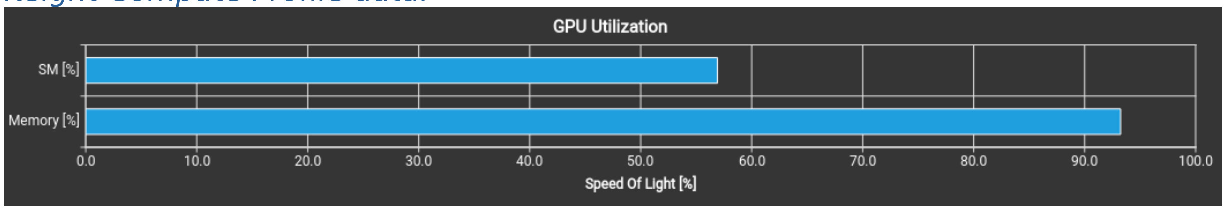
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*.231634 ms*	*.83974 ms*	*0m 1.284s*	*.86*
1000	*2.14323 ms*	*8.19904 ms*	*0m 10.354s*	*.886*
10000	*21.4202 ms*	*81.7411 ms*	*1m 36.428s*	*.8714*

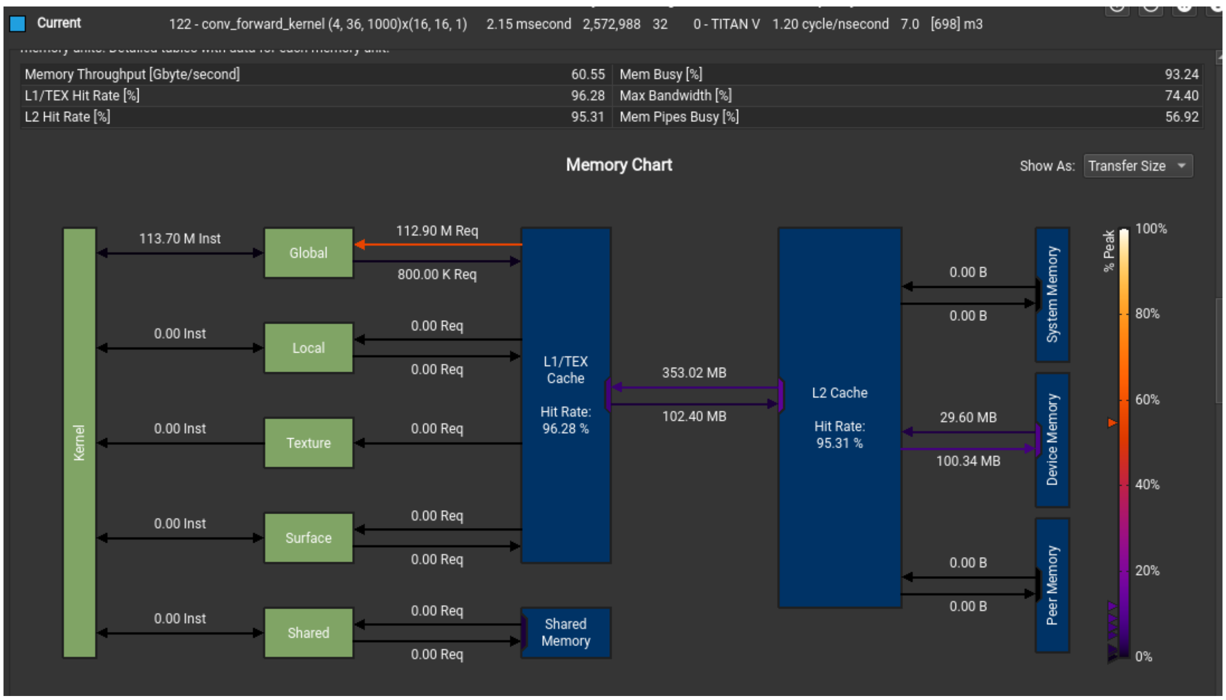
d. Was implementing this optimization successful in improving performance? Why or why not?
nsys Profile data:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   76.8      1061431558           8     132678944.8           19519       574981321  cudaMemcpy
   14.3       198136198           8      24767024.7           72197       193575437  cudaMalloc
    7.5       103479856           6      17246642.7            3029        81952365  cudaDeviceSynchronize
    1.2        16945637           6       2824272.8           15616        16830797  cudaLaunchKernel
    0.2         2759998           8        344999.8           64750          931072  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0       103452502           2      51726251.0        21501633        81950869  conv_forward_kernel
    0.0            2752           2          1376.0            1344            1408  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1280            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   91.5       960951581           2     480475790.5       386771187       574180394  [CUDA memcpy DtoH]
    8.5        89527364           6      14921227.3            1216        47985359  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
```

According to the data above for the CUDA API calls there is over a 5% increase in cudaMemcpy from 71.6% (Milestone 2) to 76.8% and a decrease in cudaDeviceSynchronize calls from 11.2% (also Milestone 2) to 7.5%, meaning the parallelism in the code increased as there was less need for cudaDeviceSynchronize and additionally due to the reduced memory access latency and utilization of the cache the cudaMemcpy calls went up. This optimization is thus a success in improving performance as the previously mentioned advantages helped to achieve an OP time of approximately 103 ms.

Nsight-Compute Profile data:





This optimization is an improvement over the baseline because we can see that uses approximately 57% of SM and 93% of memory compared to approximately 75% and 60% from Milestone 2. The L1 Cache and L2 Cache are also approximately 96% and 95% meaning that this optimization is a boost in performance as it reduces memory access latenc

e. Modified code:

```
// Insert your GPU convolution kernel code here
int m = blockIdx.x;
int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
int b = blockIdx.z;

float acc = 0.0f;
for(int c = 0; c < Channel; c++) { // sum over all input ch
    acc += in_4d(b,c,h+0,w+0) * mask_4d(m,c,0,0);
    acc += in_4d(b,c,h+0,w+1) * mask_4d(m,c,0,1);
    acc += in_4d(b,c,h+0,w+2) * mask_4d(m,c,0,2);
    acc += in_4d(b,c,h+0,w+3) * mask_4d(m,c,0,3);
    acc += in_4d(b,c,h+0,w+4) * mask_4d(m,c,0,4);
    acc += in_4d(b,c,h+0,w+5) * mask_4d(m,c,0,5);
    acc += in_4d(b,c,h+0,w+6) * mask_4d(m,c,0,6);
    acc += in_4d(b,c,h+1,w+0) * mask_4d(m,c,1,0);
    acc += in_4d(b,c,h+1,w+1) * mask_4d(m,c,1,1);
    acc += in_4d(b,c,h+1,w+2) * mask_4d(m,c,1,2);
    acc += in_4d(b,c,h+1,w+3) * mask_4d(m,c,1,3);
    acc += in_4d(b,c,h+1,w+4) * mask_4d(m,c,1,4);
    acc += in_4d(b,c,h+1,w+5) * mask_4d(m,c,1,5);
    acc += in_4d(b,c,h+1,w+6) * mask_4d(m,c,1,6);
    acc += in_4d(b,c,h+2,w+0) * mask_4d(m,c,2,0);
    acc += in_4d(b,c,h+2,w+1) * mask_4d(m,c,2,1);
    acc += in_4d(b,c,h+2,w+2) * mask_4d(m,c,2,2);
    acc += in_4d(b,c,h+2,w+3) * mask_4d(m,c,2,3);
    acc += in_4d(b,c,h+2,w+4) * mask_4d(m,c,2,4);
    acc += in_4d(b,c,h+2,w+5) * mask_4d(m,c,2,5);
    acc += in_4d(b,c,h+2,w+6) * mask_4d(m,c,2,6);
    acc += in_4d(b,c,h+3,w+0) * mask_4d(m,c,3,0);
    acc += in_4d(b,c,h+3,w+1) * mask_4d(m,c,3,1);
    acc += in_4d(b,c,h+3,w+2) * mask_4d(m,c,3,2);
    acc += in_4d(b,c,h+3,w+3) * mask_4d(m,c,3,3);
    acc += in_4d(b,c,h+3,w+4) * mask_4d(m,c,3,4);
    acc += in_4d(b,c,h+3,w+5) * mask_4d(m,c,3,5);
    acc += in_4d(b,c,h+3,w+6) * mask_4d(m,c,3,6);
    acc += in_4d(b,c,h+4,w+0) * mask_4d(m,c,4,0);
    acc += in_4d(b,c,h+4,w+1) * mask_4d(m,c,4,1);
    acc += in_4d(b,c,h+4,w+2) * mask_4d(m,c,4,2);
    acc += in_4d(b,c,h+4,w+3) * mask_4d(m,c,4,3);
    acc += in_4d(b,c,h+4,w+4) * mask_4d(m,c,4,4);
    acc += in_4d(b,c,h+4,w+5) * mask_4d(m,c,4,5);
    acc += in_4d(b,c,h+4,w+6) * mask_4d(m,c,4,6);
    acc += in_4d(b,c,h+5,w+0) * mask_4d(m,c,5,0);
    acc += in_4d(b,c,h+5,w+1) * mask_4d(m,c,5,1);
```

4. Using Streams to overlap computation with data transfer

a. Which optimization did you choose to implement? Explain why did you choose that optimization technique.
I chose the optimization technique of implementing streams to overlap computation with data transfer because I thought it would make the code more seamless by reducing the bottleneck of time required for data transferred and that the streams would synergize with each other in order to further the performance.

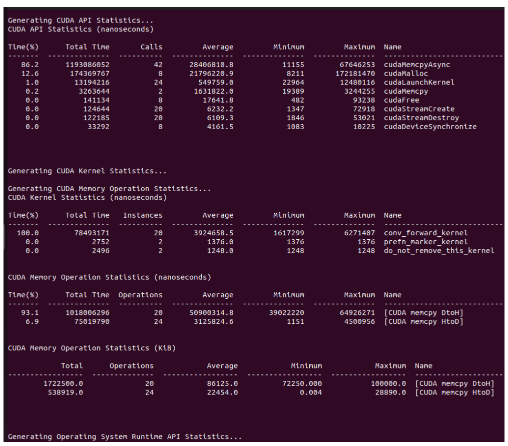
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?
This optimization works by dividing the device resources into different streams which are all tasked with different operations such as data transfer or convolution. These streams can work hand in hand with one another to help one another complete their tasks and this significantly cuts down on the time needed for data transfer as it

is done concurrently with the convolution. As a result of the multifaceted capabilities of streams, the performance is enhanced by cutting down on time and increasing parallelism. This optimization technique can synergize with techniques like memory pooling or batching to reuse device memory for operations.

c. Batch profiling results:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*.000685 ms*	*.001195 ms*	*0m 1.231s*	*.86*
1000	*.000758 ms*	*.000903 ms*	*0m 11.159s*	*.886*
10000	*.00071 ms*	*.001263 ms*	*1m 48.981s*	*.8714*

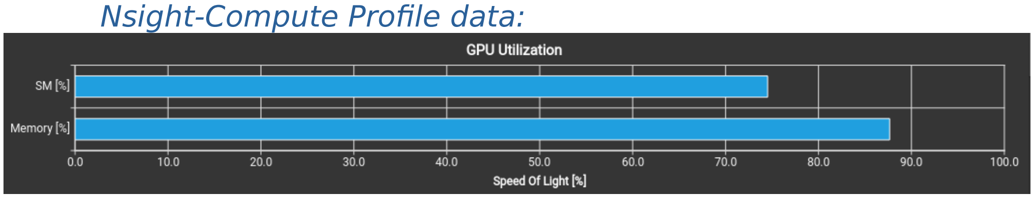
d. Was implementing this optimization successful in improving performance? Why or why not?

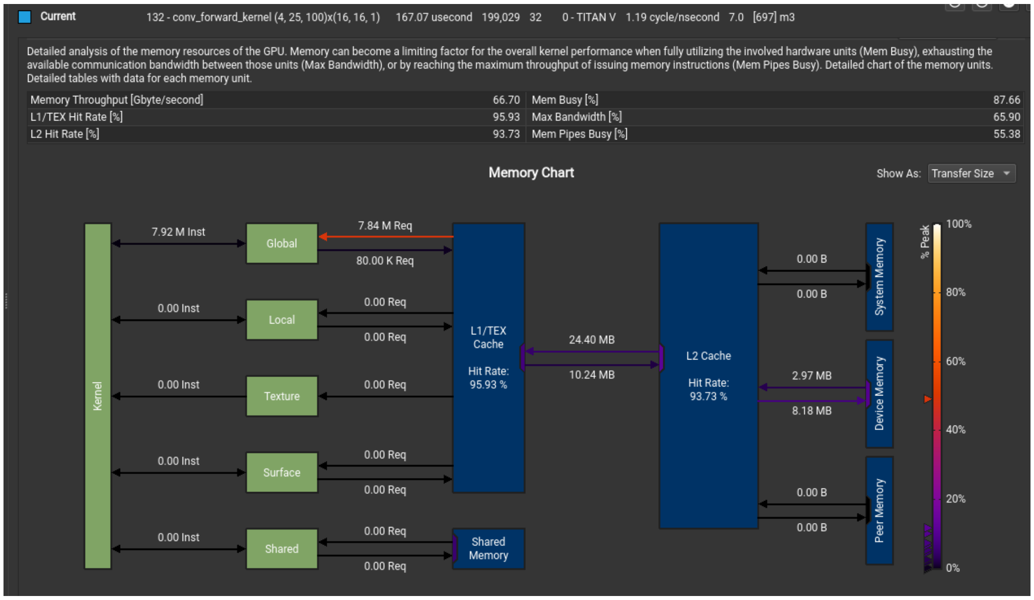


```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   86.2      1193086052          42      28406810.8           11155        67646253  cudaMemcpyAsync
   12.6       174369767           8      21796220.9            8211       172181470  cudaMalloc
    1.0        13194216          24        549759.0           22964        12480116  cudaLaunchKernel
    0.2         3263644           2       1631822.0           19389         3244255  cudaMemcpy
    0.0          141134           8         17641.8             482           93238  cudaFree
    0.0          124644          20          6232.2            1347           72918  cudaStreamCreate
    0.0          122185          20          6109.3            1846           53021  cudaStreamDestroy
    0.0           33292           8          4161.5            1083           10225  cudaDeviceSynchronize




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        78493171          20       3924658.5         1617299         6271407  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  prefn_marker_kernel
    0.0            2496           2          1248.0            1248            1248  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   93.1      1018006296          20      50900314.8        39022220        64926271  [CUDA memcpy DtoH]
    6.9        75019790          24       3125824.6            1151         4500956  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
        1722500.0              20            86125.0          72250.000           100000.0  [CUDA memcpy DtoH]
         538919.0              24            22454.0              0.004            28890.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
```

Based on the CUDA API Statistics it seems that clearly cudaMemcpyAsync is called the most for 86.2% of the time compared to cudaMemcpy being called for 71.6% of the time in Milestone 2. This change can be attributed to the fact that we are utilizing streams to simultaneously transfer data and compute the convolution, vastly enhancing the performance time. The only potential cause for concern though is that the OP times are in the ~.001 ms for this optimization method meaning that although this was successful we are unsure as to what extent the performance was enhanced.

Nsight-Compute Profile data:





For this implementation of stream optimization, the SM and Memory usages are very high because the streams handle both the convolution and data transfer of the kernel, meaning that the code is an improvement due to it utilizing the hardware resources of the GPU to their fullest to achieve the kernel's task. The only underlying issue is still the fact that the OP times are somewhat suspect meaning that there must be underlying data points that need to be measured that I have not found on Nsight_Compute.

e. Modified code:

In conv_forward_gpu_prolog:

```
const int Height_out = Height - K + 1;
const int Width_out = Width - K + 1;
float* temp_host_output = (float*)host_output;
int STREAM_X = (Batch*Channel*Height*Width)/10;
int STREAM_Y = (Batch*Map_out*Height_out*Width_out)/10;
int W_grid = (Width_out+TILE_WIDTH-1)/TILE_WIDTH;
int H_grid = (Height_out+TILE_WIDTH-1)/TILE_WIDTH;
int Y_grid = W_grid * H_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(Map_out, Y_grid, Batch/10);
cudaMalloc((void **) device_output_ptr, Batch*Map_out*Height_out*Width_out*sizeof(float));
cudaMalloc((void **) device_input_ptr, Batch*Channel*Height*Width*sizeof(float));
cudaMalloc((void **) device_mask_ptr, Map_out*Channel*K*K*sizeof(float));

cudaStream_t stream[10];
for(int i = 0; i < 10; i++)
    cudaStreamCreate(&stream[i]);
cudaMemcpyAsync(*device_mask_ptr, host_mask, Map_out*Channel*K*K*sizeof(float), cudaMemcpyHostToDevice, stream[0]);
for(int i = 0; i < 10; i++){
    int x = STREAM_X*i;
    int y = STREAM_Y*i;
    cudaMemcpyAsync((*device_input_ptr)+x, host_input+x, STREAM_X*sizeof(float), cudaMemcpyHostToDevice, stream[i]);
    conv_forward_kernel<<<gridDim, blockDim, 0, stream[i]>>>((*device_output_ptr)+y, (*device_input_ptr)+x, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);
    cudaMemcpyAsync(temp_host_output+y, (*device_output_ptr)+y, STREAM_Y*sizeof(float), cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();
cudaStreamDestroy(stream[0]);
cudaStreamDestroy(stream[1]);
cudaStreamDestroy(stream[2]);
cudaStreamDestroy(stream[3]);
cudaStreamDestroy(stream[4]);
cudaStreamDestroy(stream[5]);
cudaStreamDestroy(stream[6]);
cudaStreamDestroy(stream[7]);
cudaStreamDestroy(stream[8]);
cudaStreamDestroy(stream[9]);

cudaFree(device_output_ptr);
cudaFree(device_input_ptr);
cudaFree(device_mask_ptr);
```