Name: *Dongming Liu*
NetID: *dl35*
Section: *AL*

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	*0.175719 ms*	*0.633705 ms*	*0m1.220s*	*0.86*
1000	*1.59055 ms*	*6.10214 ms*	*0m9.758s*	*0.886*
10000	*15.7009 ms*	*60.7863 ms*	*1m35.228s*	*0.8714*

1. **Optimization 1: Weight matrix in constant memory (1 points)**

   a. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

      ☐Tiled shared memory convolution (**2 points**)
      ☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
      ☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
      ☒ Weight matrix in constant memory (**1 point**)
      ☐ Tuning with restrict and loop unrolling (**3 points**)
      ☐ Sweeping various parameters to find best values (**1 point**)
      ☐ Multiple kernel implementations for different layer sizes (**1 point**)
      ☐ Input channel reduction: tree (**3 point**)
      ☐ Input channel reduction: atomics (**2 point**)
      ☐ Fixed point (FP16) arithmetic. (**4 points**)
      ☐ Using Streams to overlap computation with data transfer (**4 points**)
      ☐ An advanced matrix multiplication algorithm (**5 points**)
      ☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
      ☐ Overlap-Add method for FFT-based convolution (**8 points**)
      ☐ Other optimizations: please explain

      I select weight matrix in constant memory as one of my optimizations. I picked this one because it is very straightforward, and this technique appears in one of the previous MPs.

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?

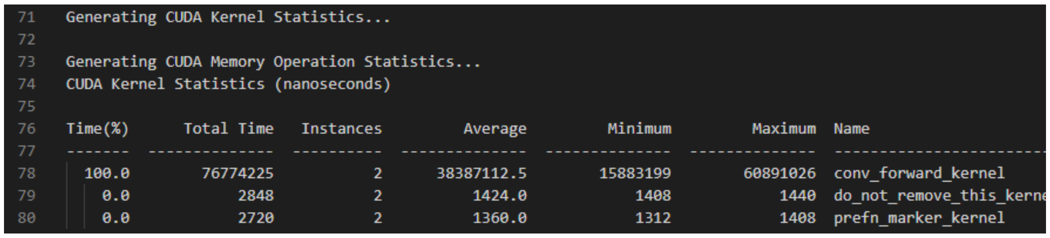
   The optimization works by transferring the weight mask to the constant memory and reducing global memory access. Since the convolution kernel never updates the mask, it is better to store the weight mask on constant memory. I thought that this technique would increase the performance of the forward convolution. Constant memory is faster than global memory in general, so using constant memory can reduce global memory access. This is the first optimization.

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	*0.166936*	*0.570333 ms*	*0m1.204s*	*0.86*
1000	*1.47756 ms*	*5.6494 ms*	*0m9.715s*	*0.886*
10000	*14.6025 ms*	*56.799 ms*	*1m35.093s*	*0.8714*

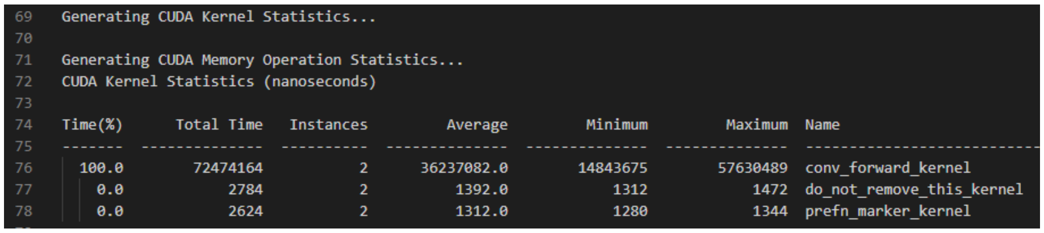
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of)

   The optimization was successful in improving performance.



```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------   --------------  ----------  --------------  --------------  --------------  -----------------------
  100.0        76774225           2      38387112.5        15883199        60891026  conv_forward_kernel
    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kerne
    0.0            2720           2          1360.0            1312            1408  prefn_marker_kernel
```

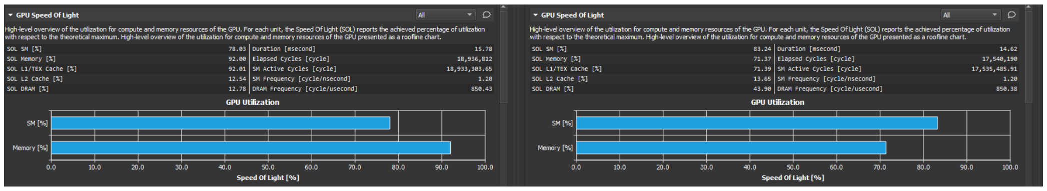
Kernel statistics in m2 implementation



```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------   --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        72474164           2      36237082.0        14843675        57630489  conv_forward_kernel
    0.0            2784           2          1392.0            1312            1472  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```

Kernel statistics in optimization 1

Based on the result from nsys, the conv_forward_kernel total time of optimization 1 is 72.47 ms in total which is less than that of m2 implementation, 76.77 ms.



```
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 [%]                 78.03 | Duration [msecond]                  15.78
SOL Memory [%]             92.00 | Elapsed Cycles [cycle]         18,936,812
SOL L1/TEX Cache [%]       92.01 | SM Active Cycles [cycle]    18,933,303.65
SOL L2 Cache [%]           12.54 | SM Frequency [cycle/nsecond]         1.20
SOL DRAM [%]               12.78 | DRAM Frequency [cycle/usecond]     850.43
```

```
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 [%]                 83.24 | Duration [msecond]                  14.62
SOL Memory [%]             71.37 | Elapsed Cycles [cycle]         17,540,190
SOL L1/TEX Cache [%]       71.39 | SM Active Cycles [cycle]    17,535,485.91
SOL L2 Cache [%]           13.65 | SM Frequency [cycle/nsecond]         1.20
SOL DRAM [%]               43.90 | DRAM Frequency [cycle/usecond]     850.38
```

Left: m2 implementation, right: optimization 1

The SM utilization of optimization 1 is also higher than the SM utilization in m2. This indicates that optimization 1 can use SM more efficiently to compute the convolution output.

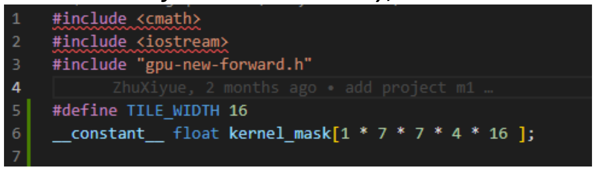
e. What references did you use when implementing this technique?

MP4: 3D convolution
Lectures: lecture 7 and lecture 8

f. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.
For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling

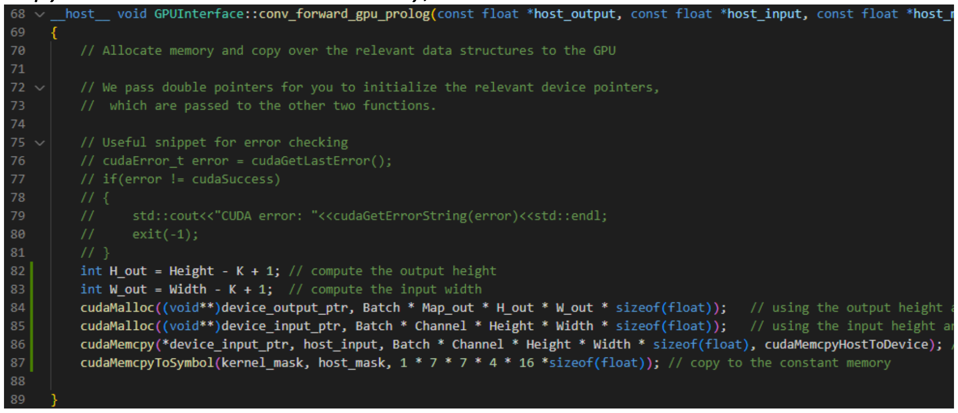
Code can be found under a folder called optimize in code submission.

Declaration of constant memory, line 6



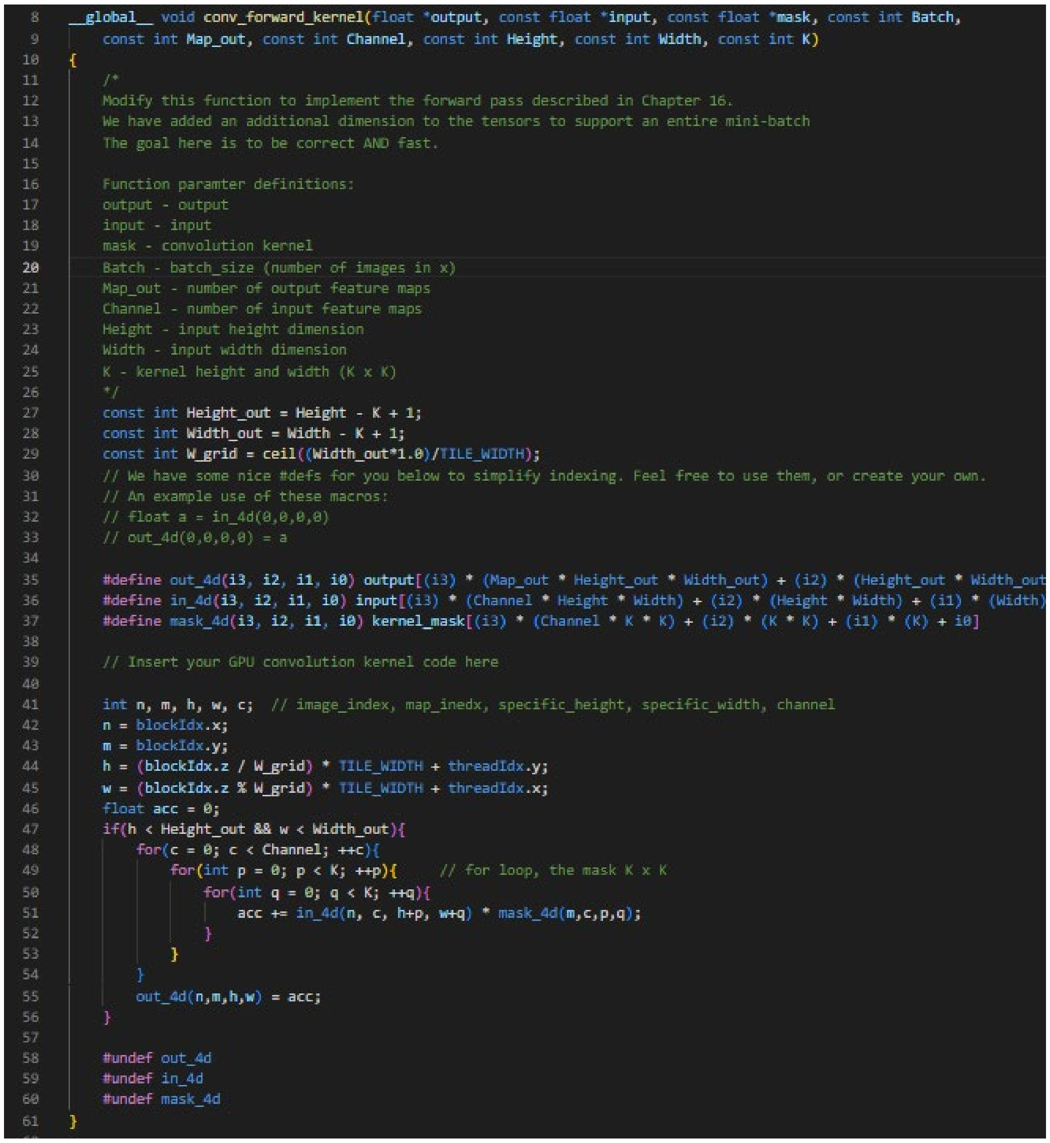
```
1  #include <cmath>
2  #include <iostream>
3  #include "gpu-new-forward.h"
4          ZhuXiyue, 2 months ago • add project m1 …
5  #define TILE_WIDTH 16
6  __constant__ float kernel_mask[1 * 7 * 7 * 4 * 16 ];
7
```

Copy the mask to constant memory, line 87



```
68  __host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input, const float *host_
69  {
70      // Allocate memory and copy over the relevant data structures to the GPU
71
72      // We pass double pointers for you to initialize the relevant device pointers,
73      //  which are passed to the other two functions.
74
75      // Useful snippet for error checking
76      // cudaError_t error = cudaGetLastError();
77      // if(error != cudaSuccess)
78      // {
79      //     std::cout<<"CUDA error: "<<cudaGetErrorString(error)<<std::endl;
80      //     exit(-1);
81      // }
82      int H_out = Height - K + 1; // compute the output height
83      int W_out = Width - K + 1;  // compute the input width
84      cudaMalloc((void**)device_output_ptr, Batch * Map_out * H_out * W_out * sizeof(float));   // using the output height a
85      cudaMalloc((void**)device_input_ptr, Batch * Channel * Height * Width * sizeof(float));   // using the input height ar
86      cudaMemcpy(*device_input_ptr, host_input, Batch * Channel * Height * Width * sizeof(float), cudaMemcpyHostToDevice); /
87      cudaMemcpyToSymbol(kernel_mask, host_mask, 1 * 7 * 7 * 4 * 16 *sizeof(float)); // copy to the constant memory
88
89  }
```

Kernel code



```
__global__ void conv_forward_kernel(float *output, const float *input, const float *mask, const int Batch,
    const int Map_out, const int Channel, const int Height, const int Width, const int K)
{
    /*
    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 = ceil((Width_out*1.0)/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
    #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * Width) + (i1) * (Width)
    #define mask_4d(i3, i2, i1, i0) kernel_mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here

    int n, m, h, w, c;  // image_index, map_inedx, specific_height, specific_width, channel
    n = blockIdx.x;
    m = blockIdx.y;
    h = (blockIdx.z / W_grid) * TILE_WIDTH + threadIdx.y;
    w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0;
    if(h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c){
            for(int p = 0; p < K; ++p){     // for loop, the mask K x K
                for(int q = 0; q < K; ++q){
                    acc += in_4d(n, c, h+p, w+q) * mask_4d(m,c,p,q);
                }
            }
        }
        out_4d(n,m,h,w) = acc;
    }

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

2. **Optimization 2: tile shared memory convolution (2 points)**

   a. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

      ☒Tiled shared memory convolution (**2 points**)
      ☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
      ☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
      ☐ Weight matrix in constant memory (**1 point**)
      ☐ Tuning with restrict and loop unrolling (**3 points**)

☐ Sweeping various parameters to find best values (**1 point**)
☐ Multiple kernel implementations for different layer sizes (**1 point**)
☐ Input channel reduction: tree (**3 point**)
☐ Input channel reduction: atomics (**2 point**)
☐ Fixed point (FP16) arithmetic. (**4 points**)
☐ Using Streams to overlap computation with data transfer (**4 points**)
☐ An advanced matrix multiplication algorithm (**5 points**)
☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
☐ Overlap-Add method for FFT-based convolution (**8 points**)
☐ Other optimizations: please explain

I chose Tile shared memory convolution because it appeared on one of the previous MPs. In addition, previous lectures talked about this technique. I thought it was easy to implement.

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?

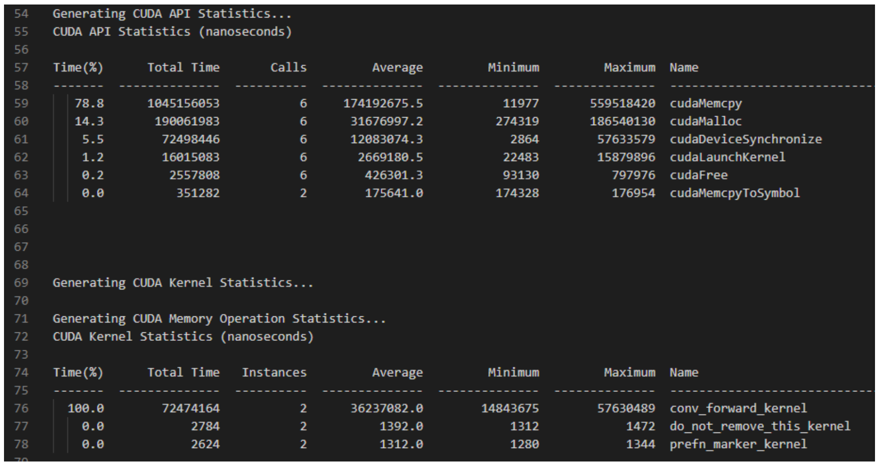
Optimization 2 works by preloading data from global memory to shared memory. During the calculations of the convolution output, the threads can directly retrieve the data from shared memory rather than from the global memory. I thought this optimization would increase the performance of the forward convolution because shared memory is much faster than global memory. This can reduce global memory access and optimize memory utilization. This optimization synergizes with optimization 1: Weight Matrix in constant memory.

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	*0.164547 ms*	*0.657649 ms*	*0m1.244s*	*0.86*
1000	*1.55923 ms*	*6.53511 ms*	*0m9.735s*	*0.886*
10000	*15.415 ms*	*64.9177 ms*	*1m34.864s*	*0.8714*

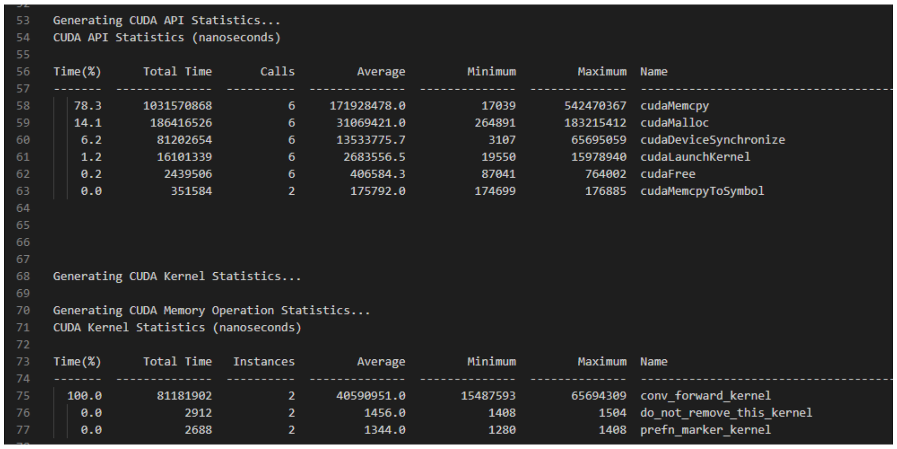
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of

The optimization was not successful in improving performance.



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  ---------------------------
   78.8     1045156053           6    174192675.5          11977      559518420  cudaMemcpy
   14.3      190061983           6     31676997.2         274319      186540130  cudaMalloc
    5.5       72498446           6     12083074.3           2864       57633579  cudaDeviceSynchronize
    1.2       16015083           6      2669180.5          22483       15879896  cudaLaunchKernel
    0.2        2557808           6       426301.3          93130         797976  cudaFree
    0.0         351282           2       175641.0         174328         176954  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  ---------------------------
  100.0       72474164           2     36237082.0       14843675       57630489  conv_forward_kernel
    0.0           2784           2         1392.0           1312           1472  do_not_remove_this_kernel
    0.0           2624           2         1312.0           1280           1344  prefn_marker_kernel
```

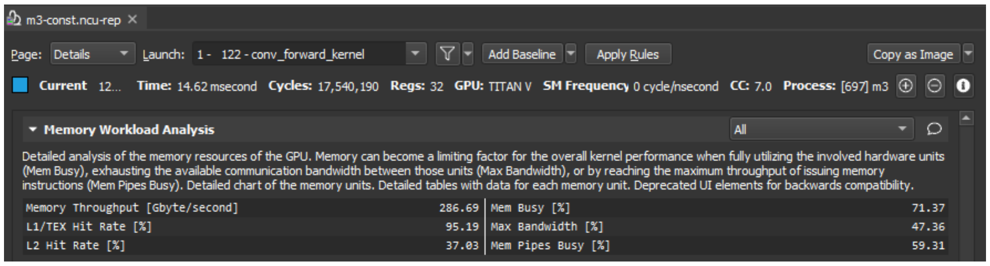
Optimization 1 on baseline m2, nsys statistics



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  -----------------------------------
   78.3     1031570868           6    171928478.0          17039      542470367  cudaMemcpy
   14.1      186416526           6     31069421.0         264891      183215412  cudaMalloc
    6.2       81202654           6     13533775.7           3107       65695059  cudaDeviceSynchronize
    1.2       16101339           6      2683556.5          19550       15978940  cudaLaunchKernel
    0.2        2439506           6       406584.3          87041         764002  cudaFree
    0.0         351584           2       175792.0         174699         176885  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  -----------------------------------
  100.0       81181902           2     40590951.0       15487593       65694309  conv_forward_kernel
    0.0           2912           2         1456.0           1408           1504  do_not_remove_this_kernel
    0.0           2688           2         1344.0           1280           1408  prefn_marker_kernel
```

Optimization 1 & 2 on baseline m2, nsys statistics

Compared with optimization 1, optimization 1&2 spent less time on cudaMemcpy and cudaMalloc. However, the total time of conv_foward_kernel in optimization 1&2 is higher than that of conv_forward_kernel in optimization 1.



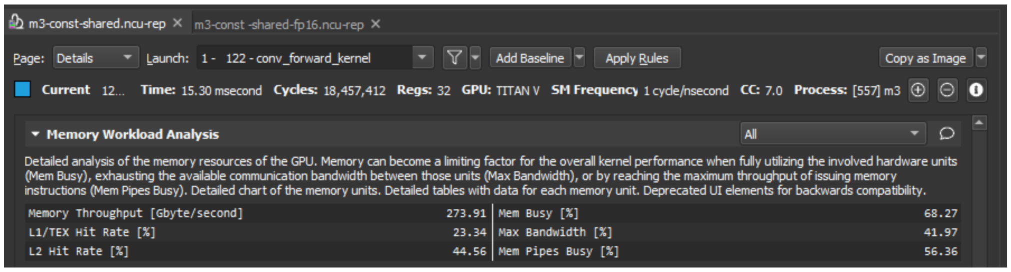
Current 12... Time: 14.62 msecond Cycles: 17,540,190 Regs: 32 GPU: TITAN V SM Frequency 0 cycle/nsecond CC: 7.0 Process: [697] m3

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit. Deprecated UI elements for backwards compatibility.

Memory Throughput [Gbyte/second]	286.69	Mem Busy [%]	71.37
L1/TEX Hit Rate [%]	95.19	Max Bandwidth [%]	47.36
L2 Hit Rate [%]	37.03	Mem Pipes Busy [%]	59.31

Optimiztion 1 on baseline m2, memory statistics



Current 12... Time: 15.30 msecond Cycles: 18,457,412 Regs: 32 GPU: TITAN V SM Frequency 1 cycle/nsecond CC: 7.0 Process: [557] m3

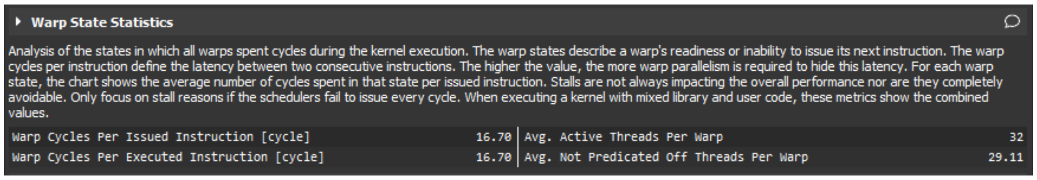
Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit. Deprecated UI elements for backwards compatibility.

```
Memory Throughput [Gbyte/second]      273.91 | Mem Busy [%]         68.27
L1/TEX Hit Rate [%]                    23.34 | Max Bandwidth [%]    41.97
L2 Hit Rate [%]                        44.56 | Mem Pipes Busy [%]   56.36
```

Optimization 1 & 2 on baseline: memory statistics

Compared with Optimization 1, the memory throughput in optimization 1&2 (273.91 GB/s) is lower than that of optimization 1 (286.69 GB/s). Even though optimization 2 has a higher L2 Hit Rate (44.56%), it has a lower L1 Hit Rate (23.34%). This is one factor that this optimization did not have a great performance.

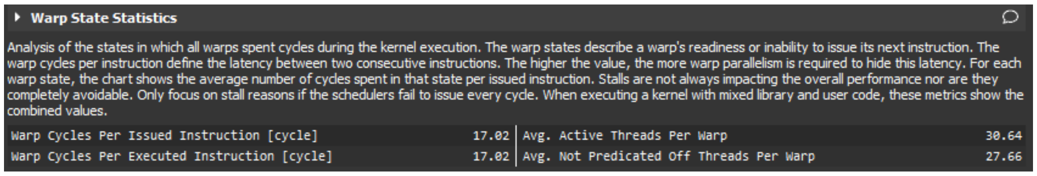


Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

```
Warp Cycles Per Issued Instruction [cycle]     16.70 | Avg. Active Threads Per Warp              32
Warp Cycles Per Executed Instruction [cycle]   16.70 | Avg. Not Predicated Off Threads Per Warp  29.11
```

Optimization 1 on baseline: warp statistics



Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

```
Warp Cycles Per Issued Instruction [cycle]     17.02 | Avg. Active Threads Per Warp              30.64
Warp Cycles Per Executed Instruction [cycle]   17.02 | Avg. Not Predicated Off Threads Per Warp  27.66
```

Optimization 1&2 on baseline: warp statistics

Compared with optimization 1, optimization 1 & 2 has fewer average active threads per warp. Optimization 1 has an average of 32, and optimization 2 has an average of 30.64. The warp cycles per issued/executed instruction in optimization 2 are not as good as the ones in optimization 2. The poor performance can be caused by the kernel code that transfers data from global memory to shared memory. Not all threads in a warp would transfer the data to the shared memory, which introduced another set of control divergences to my kernel and lowered the number of active threads. This would also bring down the performance.

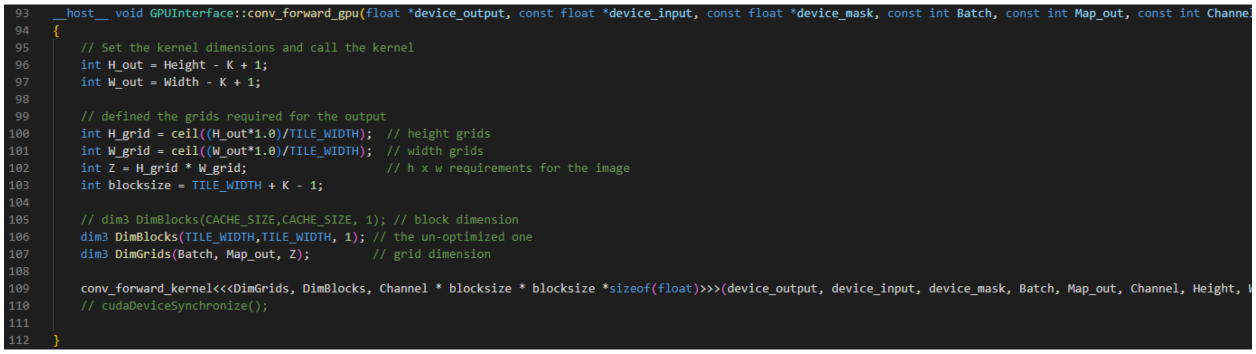
e. What references did you use when implementing this technique?

MP4: 3D convolution
Lectures: lecture 7 and lecture 8
Using Shared Memory in CUDA C/C++: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

f. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.
For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling

Code can be found under a folder called optimize in code submission.

Dynamically shared memory allocation, line 109.



```
93  __host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input, const float *device_mask, const int Batch, const int Map_out, const int Channe
94  {
95      // Set the kernel dimensions and call the kernel
96      int H_out = Height - K + 1;
97      int W_out = Width - K + 1;
98
99      // defined the grids required for the output
100     int H_grid = ceil((H_out*1.0)/TILE_WIDTH);  // height grids
101     int W_grid = ceil((W_out*1.0)/TILE_WIDTH);  // width grids
102     int Z = H_grid * W_grid;                    // h x w requirements for the image
103     int blocksize = TILE_WIDTH + K - 1;
104
105     // dim3 DimBlocks(CACHE_SIZE,CACHE_SIZE, 1); // block dimension
106     dim3 DimBlocks(TILE_WIDTH,TILE_WIDTH, 1); // the un-optimized one
107     dim3 DimGrids(Batch, Map_out, Z);         // grid dimension
108
109     conv_forward_kernel<<<DimGrids, DimBlocks, Channel * blocksize * blocksize *sizeof(float)>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, 
110     // cudaDeviceSynchronize();
111
112 }
```

Kernel code:

```
#define TILE_WIDTH 16
__constant__ float kernel_mask[1 * 7 * 7 * 4 * 16];
__global__ void conv_forward_kernel(float *output, const float *input, const float *mask, const int Batch, const int Map_out, const int Channel, const int Height, const int Width, const int K)
{
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int W_grid = ceil((Width_out*1.0)/TILE_WIDTH);
    const int blocksize = TILE_WIDTH + K - 1;

    extern __shared__ float tileMem[]; // shared memory

    #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) kernel_mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
    #define shareMem(i2, i1, i0) tileMem[(i2)*(blocksize*blocksize) + (i1) * blocksize + (i0)] // use for 3d shared memory
    // Insert your GPU convolution kernel code here
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int c;  // image_index, map_inedx, specific_height, specific_width, channel
    int h_topleft= (blockIdx.z / W_grid) * TILE_WIDTH;
    int w_topleft = (blockIdx.z % W_grid) * TILE_WIDTH;
    int n = blockIdx.x;
    int m = blockIdx.y;
    int h = h_topleft + ty; // the output height index
    int w = w_topleft + tx; // the output width index
    float acc = 0.0;

    // with shared tiles, may be able to reduce sync_threads
    for(c = 0; c < Channel; ++c){
        for(int i = ty; i < blocksize; i+= TILE_WIDTH){
            for(int j = tx; j < blocksize; j += TILE_WIDTH){
                if(h_topleft + i < Height && w_topleft + j < Width){
                    shareMem(c,i,j) = in_4d(n, c, h_topleft + i, w_topleft + j);
                }
                else{
                    shareMem(c,i,j) = 0.0f;
                }
            }
        }
    }
    __syncthreads();
    if (h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c){
            for(int p = 0; p < K; ++p){     // for loop, the mask K x K
                for(int q = 0; q < K; ++q){
                    acc += shareMem(c ,ty+p , tx+q) * mask_4d(m,c,p,q);
                    // acc += in_4d(n, c, h+p, w+q) * mask_4d(m,c,p,q);
                }
            }
        }
        out_4d(n,m,h,w) = acc;
    }
    #undef out_4d
    #undef in_4d
    #undef mask_4d
    #undef shareMem
}
```

3. **Optimization 3: Fixed point (FP16) arithmetic (4 points)**

   a. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

      ☐Tiled shared memory convolution (**2 points**)
      ☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
      ☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
      ☐ Weight matrix in constant memory (**1 point**)
      ☐ Tuning with restrict and loop unrolling (**3 points**)
      ☐ Sweeping various parameters to find best values (**1 point**)
      ☐ Multiple kernel implementations for different layer sizes (**1 point**)
      ☐ Input channel reduction: tree (**3 point**)
      ☐ Input channel reduction: atomics (**2 point**)
      ☒ Fixed point (FP16) arithmetic. (**4 points**)
      ☐ Using Streams to overlap computation with data transfer (**4 points**)
      ☐ An advanced matrix multiplication algorithm (**5 points**)
      ☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
      ☐ Overlap-Add method for FFT-based convolution (**8 points**)
      ☐ Other optimizations: please explain

      I chose this optimization because I thought this optimization would be interesting since it would change the accuracy according to the final project GitHub. In addition, I found relevant documents on FP16 arithmetic. I thought that FP16 might have some advantage over float in calculation and size.

   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?

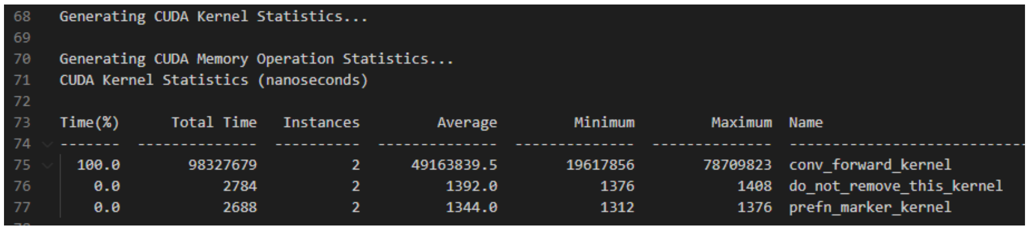
      The optimization would convert the floating points number into a new type __half (FP16). Instead of using floating point arithmetic in calculating the convolution output, the kernel calculates the convolution output with FP16 arithmetic. In the end, the result will convert back to float. I think this will improve the performance of the forward convolution. FP16 is smaller than float, which means FP16 may have a faster computation speed due to the advantage of a smaller data size. Also, FP16 requires less memory than float. I thought this optimization synergizes with optimization 1.

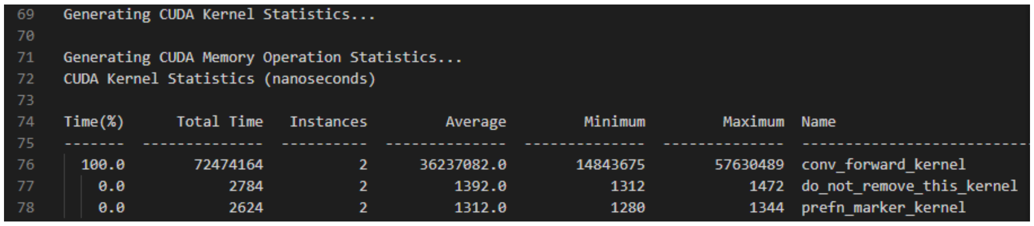
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.213373 ms*	*0.791051 ms*	*0m1.248s*	*0.86*
1000	*1.99508 ms*	*7.73802 ms*	*0m10.106s*	*0.887*
10000	*19.7247 ms*	*77.7143 ms*	*1m36.354s*	*0.8716*

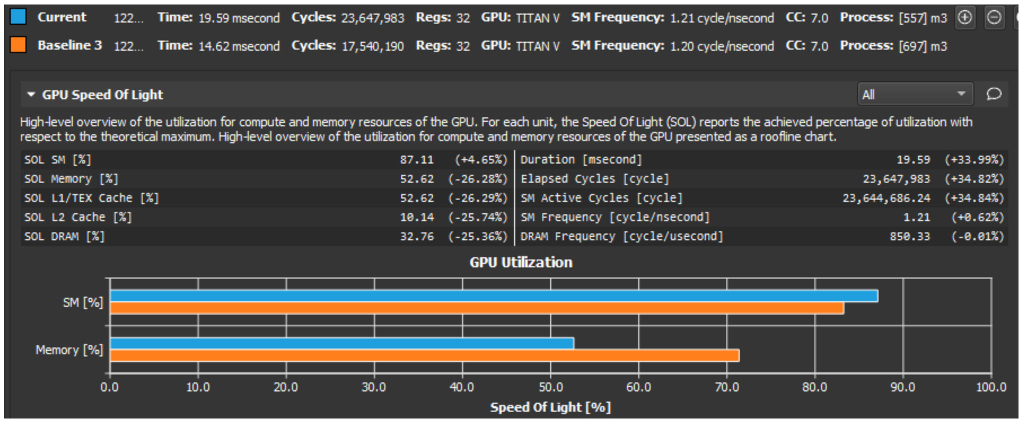
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off)

This optimization was not successful in improving performance. This optimization was built on top of optimization 1, and the Op Time for optimization 1 is much better. The accuracy was slightly higher than the accuracy in the previous implementation. This was caused by the precision of calculations and rounding in FP16.


```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------   -------------  ----------   -------------   -------------   -------------  ---------------------------
  100.0        98327679           2      49163839.5        19617856        78709823  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
```
Optimization 1 & 3 on baseline m2, took 98.33ms


```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------   -------------  ----------   -------------   -------------   -------------  ---------------------------
  100.0        72474164           2      36237082.0        14843675        57630489  conv_forward_kernel
    0.0            2784           2          1392.0            1312            1472  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```
Optimization 1 on baseline m2, took 72.47ms

Based on the data from nsys, the time that conv_forward_kernel spent in optimization 1 & 3 is larger than that of optimization 1.



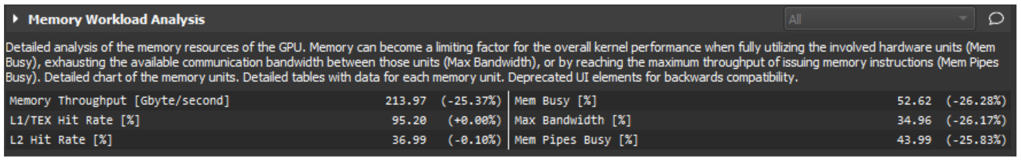
```
Current     122...  Time: 19.59 msecond  Cycles: 23,647,983  Regs: 32  GPU: TITAN V  SM Frequency: 1.21 cycle/nsecond  CC: 7.0  Process: [557] m3
Baseline 3  122...  Time: 14.62 msecond  Cycles: 17,540,190  Regs: 32  GPU: TITAN V  SM Frequency: 1.20 cycle/nsecond  CC: 7.0  Process: [697] m3
```

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 [%]                 87.11   (+4.65%) | Duration [msecond]                   19.59  (+33.99%)
SOL Memory [%]             52.62  (-26.28%) | Elapsed Cycles [cycle]          23,647,983  (+34.82%)
SOL L1/TEX Cache [%]       52.62  (-26.29%) | SM Active Cycles [cycle]     23,644,686.24  (+34.84%)
SOL L2 Cache [%]           10.14  (-25.74%) | SM Frequency [cycle/nsecond]          1.21   (+0.62%)
SOL DRAM [%]               32.76  (-25.36%) | DRAM Frequency [cycle/usecond]      850.33   (-0.01%)
```

Blue: Optimization 1 & 3 on baseline m2, Orange: Optimization 1 on baseline m2



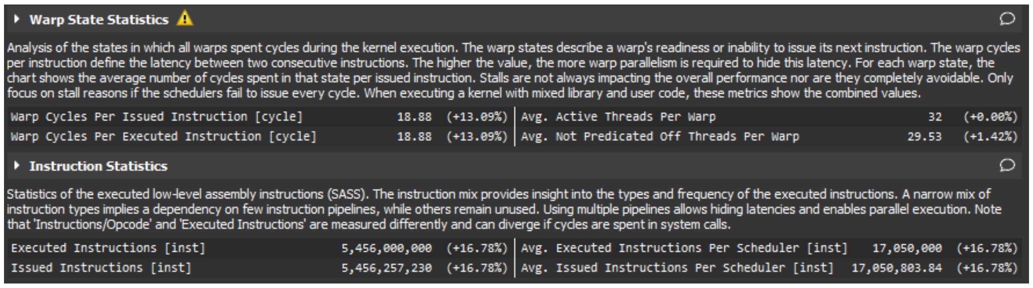
Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit. Deprecated UI elements for backwards compatibility.

```
Memory Throughput [Gbyte/second]    213.97  (-25.37%) | Mem Busy [%]          52.62  (-26.28%)
L1/TEX Hit Rate [%]                  95.20   (+0.00%) | Max Bandwidth [%]     34.96  (-26.17%)
L2 Hit Rate [%]                      36.99   (-0.10%) | Mem Pipes Busy [%]    43.99  (-25.83%)
```

Memory statistics of Optimization 1&3 on baseline m2

Based on the result from the Nsight Compute, optimization 1&3 has a lower memory utilization and higher SM utilization. This indicates that optimization 1&3 does less memory in the kernel. However, the memory throughput of optimization 1&3 is 25.37% lower than the memory throughput of optimization 1.



Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

```
Warp Cycles Per Issued Instruction [cycle]     18.88  (+13.09%) | Avg. Active Threads Per Warp               32   (+0.00%)
Warp Cycles Per Executed Instruction [cycle]   18.88  (+13.09%) | Avg. Not Predicated Off Threads Per Warp   29.53   (+1.42%)
```

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

```
Executed Instructions [inst]   5,456,000,000  (+16.78%) | Avg. Executed Instructions Per Scheduler [inst]   17,050,000  (+16.78%)
Issued Instructions [inst]     5,456,257,230  (+16.78%) | Avg. Issued Instructions Per Scheduler [inst]  17,050,803.84  (+16.78%)
```

Warp statistics and instruction statistics of Optimization 1&3 on baseline

The warp cycles per issue/execute instruction in optimization 1&3 are higher than that of optimization 1, so optimization 1&3 would have a longer instruction execution time. From the instruction statistics, the number of instructions in optimization 1&3 is 16.78% higher than the number of instructions in optimization 1. This is mainly caused by the conversion between float and FP16 (__half) data type. Although the FP16 data type may have some advantages over float type, the overheads of float to FP16 conversions and FP16 to float conversions cannot be avoided in optimization 1&3. Thus, the performance of optimization 1&3 is lower.

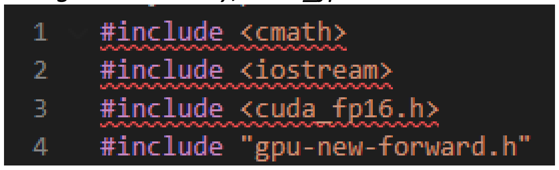
e. What references did you use when implementing this technique?

CUDA Toolkit documentation:
https://docs.nvidia.com/cuda/cuda-math-api/index.html

f. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.
For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling

Code can be found under a folder called optimize in code submission.

Using a new library, cuda_fp16.h



```
#include <cmath>
#include <iostream>
#include <cuda_fp16.h>
#include "gpu-new-forward.h"
```

Kernel code

```
#define TILE_WIDTH 16
__constant__ float kernel_mask[1 * 7 * 7 * 4 * 16];

__global__ void conv_forward_kernel(float *output, const float *input, const
float *mask, const int Batch,
    const int Map_out, const int Channel, const int Height, const int Width,
const int K)
{
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int W_grid = ceil((Width_out*1.0)/TILE_WIDTH);

    #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) kernel_mask[(i3) * (Channel * K * K) + (i2)
* (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here

    int n, m, h, w, c;  // image_index, map_inedx, specific_height,
specific_width, channel
    n = blockIdx.x;
    m = blockIdx.y;
    h = (blockIdx.z / W_grid) * TILE_WIDTH + threadIdx.y;
    w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;
    __half acc = __float2half_rn(0.0f);
    if(h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c){
            for(int p = 0; p < K; ++p){     // for loop, the mask K x K
                for(int q = 0; q < K; ++q){
                    acc = __hadd(acc, __hmul(__float2half_rn(in_4d(n, c, h+p,
w+q)), __float2half_rn(mask_4d(m,c,p,q))));
                }
            }
        }
        out_4d(n,m,h,w) = __half2float(acc);
    }

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

4. **Optimization 4: Using Streams to overlap computation with data transfer (4 point)**

   a. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

      ☐Tiled shared memory convolution (**2 points**)
      ☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
      ☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
      ☐ Weight matrix in constant memory (**1 point**)
      ☐ Tuning with restrict and loop unrolling (**3 points**)
      ☐ Sweeping various parameters to find best values (**1 point**)
      ☐ Multiple kernel implementations for different layer sizes (**1 point**)
      ☐ Input channel reduction: tree (**3 point**)
      ☐ Input channel reduction: atomics (**2 point**)
      ☐ Fixed point (FP16) arithmetic. (**4 points**)
      ☒ Using Streams to overlap computation with data transfer (**4 points**)
      ☐ An advanced matrix multiplication algorithm (**5 points**)
      ☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
      ☐ Overlap-Add method for FFT-based convolution (**8 points**)
      ☐ Other optimizations: please explain

      I choose using streams to overlap computation with data transfer because there is a lecture talk about this. Implementing streams can increase the overall performance and data transfer rate.

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?

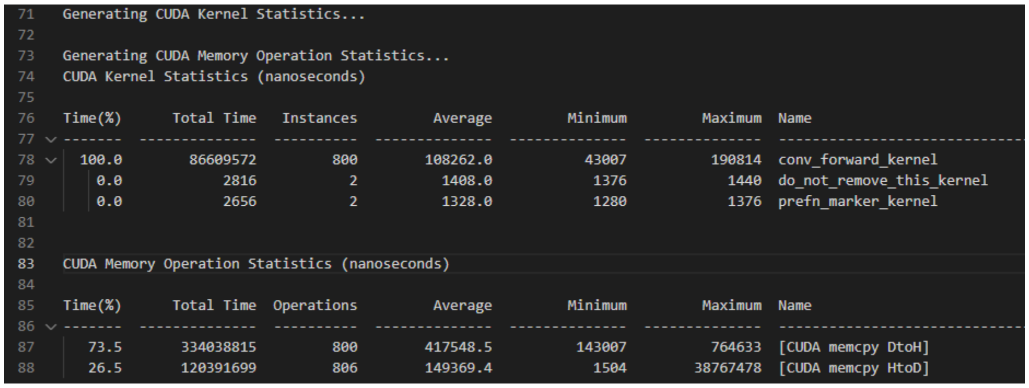
This optimization works by creating multiple streams to overlap kernel execution and data transfer. A large chunk of data is broken into smaller chunks and distributed among the streams. I thought this would increase the overall performance. Some streams might execute the kernel whereas other streams transfer the data and prepare for kernel execution. This creates parallelism among the streams and reduces a certain amount of latency in data transfer. I think the optimization synergizes with optimization 1 and produces better performance.

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	*0.003278 ms*	*0.002823 ms*	*0m1.236s*	*0.86*
1000	*0.002481 ms*	*0.003544 ms*	*0m9.889s*	*0.886*
10000	*0.003199 ms*	*0.003446 ms*	*1m38.379s*	*0.8714*

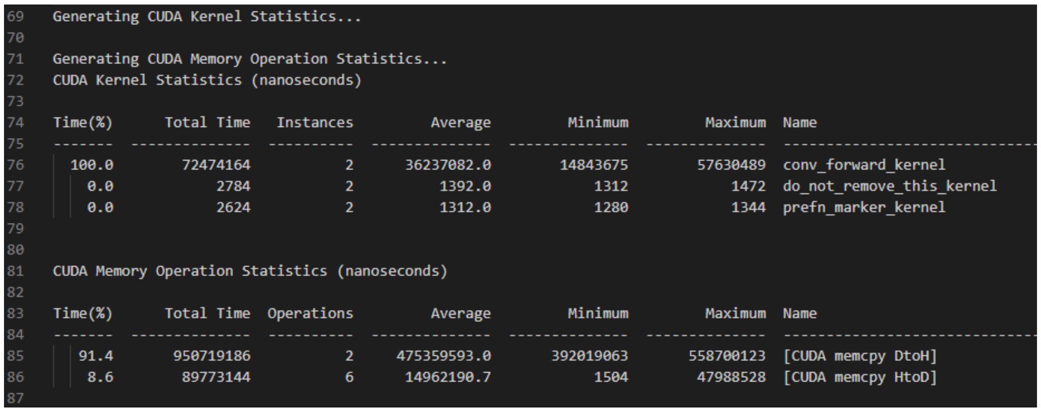
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of

The optimization was not very successful in improving performance. Based on the nsys output data below, even though optimization 1&4 have better total time in memory operation (454.43 ms) than the total time in memory operation of optimization 1 (1040.49 ms), the total time in conv_forward_kernel of optimization 1&4 (86.61 ms) did not beat the total time of optimization 1 (72.47ms).



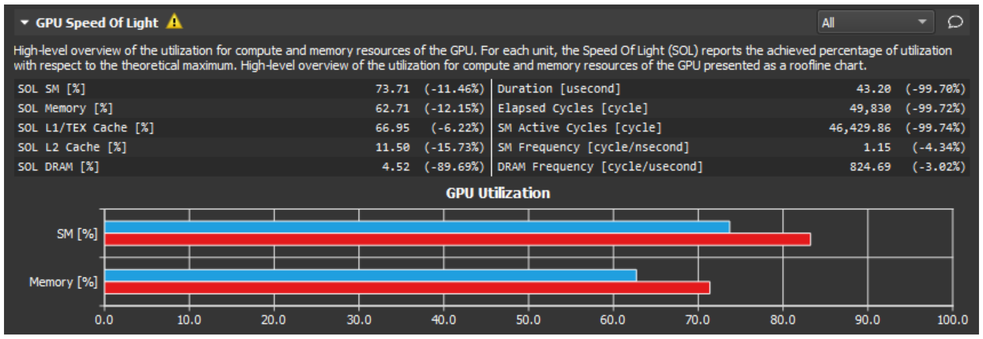
```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        86609572         800        108262.0           43007          190814  conv_forward_kernel
    0.0            2816           2          1408.0            1376            1440  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
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   73.5       334038815         800        417548.5          143007          764633  [CUDA memcpy DtoH]
   26.5       120391699         806        149369.4            1504        38767478  [CUDA memcpy HtoD]
```

Optimization 1&4 on baseline m2



```
Generating CUDA Kernel Statistics...

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

Time(%)   Total Time   Instances       Average       Minimum       Maximum  Name
-------  -----------  ----------  ------------  ------------  ------------  ----------------------------
  100.0     72474164           2    36237082.0      14843675      57630489  conv_forward_kernel
    0.0         2784           2        1392.0          1312          1472  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    950719186           2   475359593.0     392019063     558700123  [CUDA memcpy DtoH]
    8.6     89773144           6    14962190.7          1504      47988528  [CUDA memcpy HtoD]
```

Optimization 1 on baseline m2

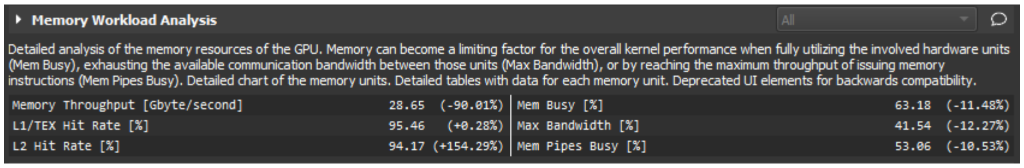


GPU Speed Of Light

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 [%]                 73.71  (-11.46%) | Duration [usecond]                    43.20  (-99.70%)
SOL Memory [%]             62.71  (-12.15%) | Elapsed Cycles [cycle]               49,830  (-99.72%)
SOL L1/TEX Cache [%]       66.95   (-6.22%) | SM Active Cycles [cycle]          46,429.86  (-99.74%)
SOL L2 Cache [%]           11.50  (-15.73%) | SM Frequency [cycle/nsecond]           1.15   (-4.34%)
SOL DRAM [%]                4.52  (-89.69%) | DRAM Frequency [cycle/usecond]       824.69   (-3.02%)
```

Blue: optimization 1&4 on baseline m2, Red: optimization 1 on baseline m2



Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit. Deprecated UI elements for backwards compatibility.

```
Memory Throughput [Gbyte/second]      28.65   (-90.01%) | Mem Busy [%]           63.18  (-11.48%)
L1/TEX Hit Rate [%]                   95.46    (+0.28%) | Max Bandwidth [%]      41.54  (-12.27%)
L2 Hit Rate [%]                       94.17  (+154.29%) | Mem Pipes Busy [%]     53.06  (-10.53%)
```

Optimization 1&4 on baseline m2 compared with Optimization 1 on m2

Based on the information from Nsight Compute, both utilization on SM and memory in optimization decreases. The memory throughput of each kernel launch in optimization 1&4 is not as good as the one in optimization 1. The good things are that the L1 and L2 hit rate in optimization 1&4 are high. The decrease in the performance may come from the number of streams that I used in my code and data access patterns. The number of streams that I picked may not be optimized one for this convolution kernel. The amount of overlapping between kernel execution and data transfer is not sufficient.

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

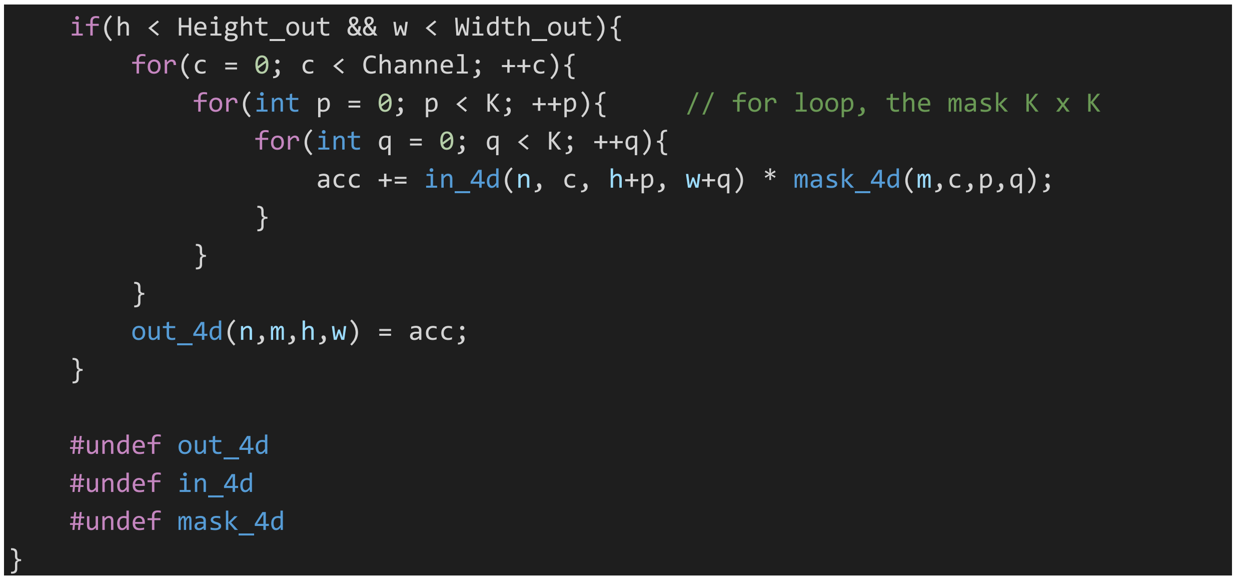
Lecture 22

f. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.
For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling

Code can be found under a folder called optimize in code submission.

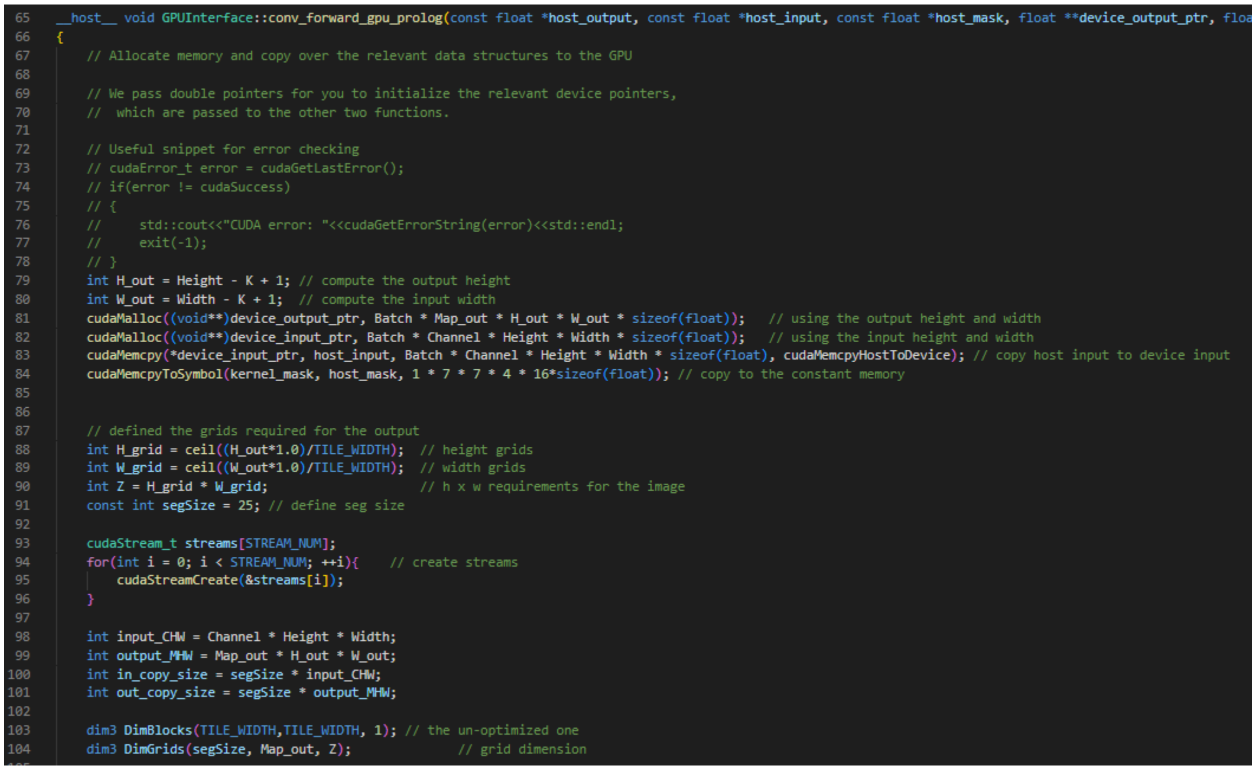
Kernel code

```
#define TILE_WIDTH 16
#define STREAM_NUM  4
__constant__ float kernel_mask[1 * 7 * 7 * 4 * 16];

__global__ void conv_forward_kernel(float *output, const float *input, const
float *mask, const int Batch,
    const int Map_out, const int Channel, const int Height, const int Width,
const int K)
{
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int W_grid = ceil((Width_out*1.0)/TILE_WIDTH);

    #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) kernel_mask[(i3) * (Channel * K * K) + (i2)
* (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here

    int n, m, h, w, c;  // image_index, map_inedx, specific_height,
specific_width, channel
    n = blockIdx.x;
    m = blockIdx.y;
    h = (blockIdx.z / W_grid) * TILE_WIDTH + threadIdx.y;
    w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0;
```



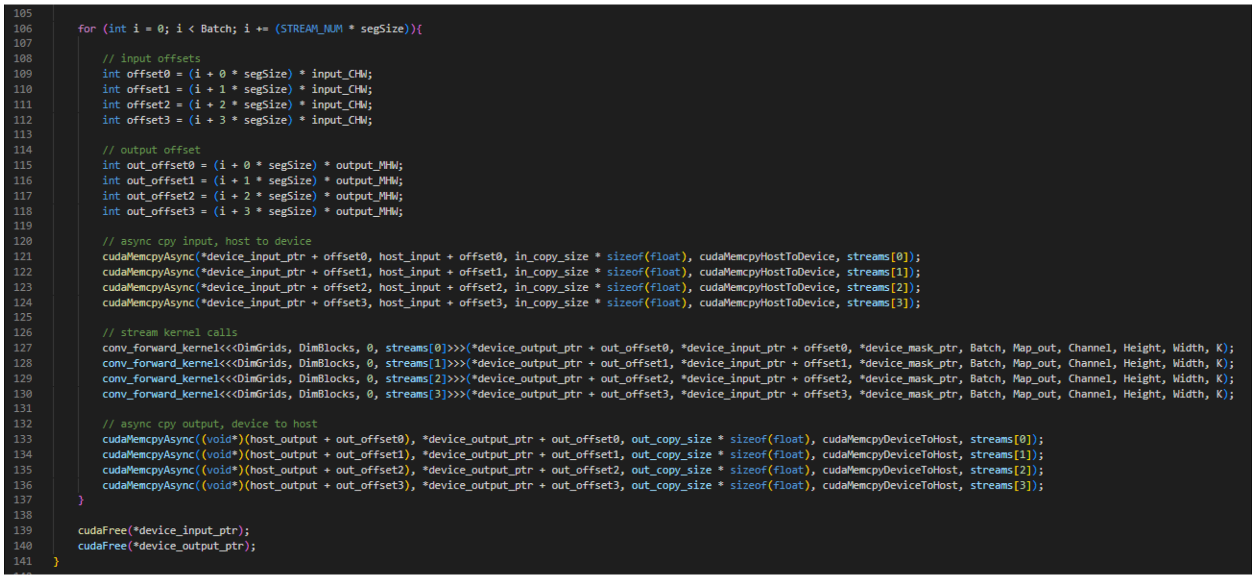
```
    if(h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c){
            for(int p = 0; p < K; ++p){     // for loop, the mask K x K
                for(int q = 0; q < K; ++q){
                    acc += in_4d(n, c, h+p, w+q) * mask_4d(m,c,p,q);
                }
            }
        }
        out_4d(n,m,h,w) = acc;
    }

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

create streams and declare some useful variables



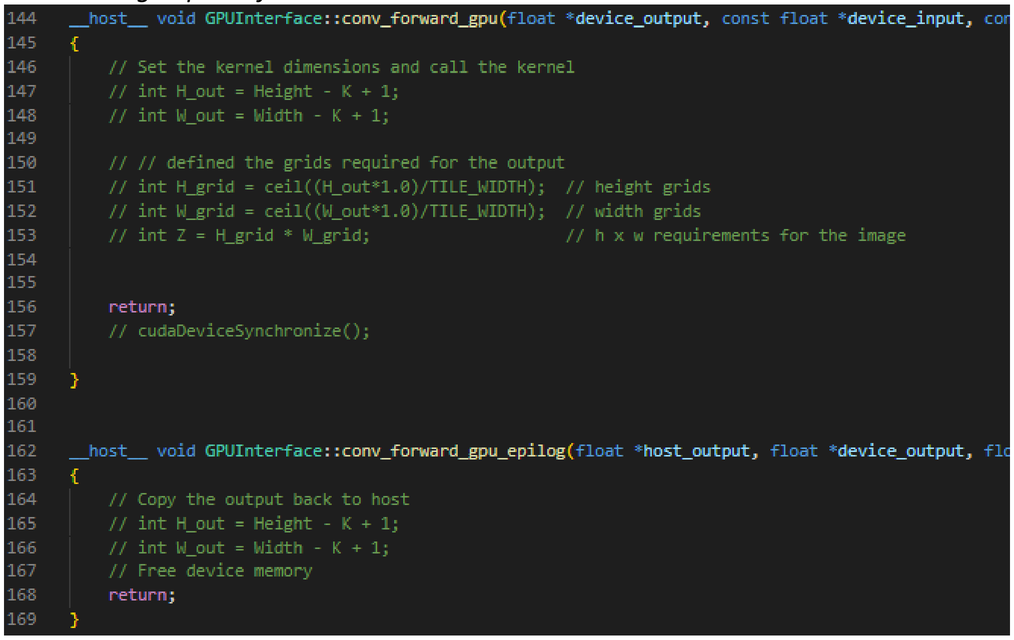
```
__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input, const float *host_mask, float **device_output_ptr, floa
{
    // Allocate memory and copy over the relevant data structures to the GPU

    // 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);
    // }
    int H_out = Height - K + 1; // compute the output height
    int W_out = Width - K + 1;  // compute the input width
    cudaMalloc((void**)device_output_ptr, Batch * Map_out * H_out * W_out * sizeof(float));   // using the output height and width
    cudaMalloc((void**)device_input_ptr, Batch * Channel * Height * Width * sizeof(float));   // using the input height and width
    cudaMemcpy(*device_input_ptr, host_input, Batch * Channel * Height * Width * sizeof(float), cudaMemcpyHostToDevice); // copy host input to device input
    cudaMemcpyToSymbol(kernel_mask, host_mask, 1 * 7 * 7 * 4 * 16*sizeof(float)); // copy to the constant memory


    // defined the grids required for the output
    int H_grid = ceil((H_out*1.0)/TILE_WIDTH);  // height grids
    int W_grid = ceil((W_out*1.0)/TILE_WIDTH);  // width grids
    int Z = H_grid * W_grid;                    // h x w requirements for the image
    const int segSize = 25; // define seg size

    cudaStream_t streams[STREAM_NUM];
    for(int i = 0; i < STREAM_NUM; ++i){    // create streams
        cudaStreamCreate(&streams[i]);
    }

    int input_CHW = Channel * Height * Width;
    int output_MHW = Map_out * H_out * W_out;
    int in_copy_size = segSize * input_CHW;
    int out_copy_size = segSize * output_MHW;

    dim3 DimBlocks(TILE_WIDTH,TILE_WIDTH, 1); // the un-optimized one
    dim3 DimGrids(segSize, Map_out, Z);              // grid dimension
```

The for loop for data transfer and kernel launch with streams



```
    for (int i = 0; i < Batch; i += (STREAM_NUM * segSize)){

        // input offsets
        int offset0 = (i + 0 * segSize) * input_CHW;
        int offset1 = (i + 1 * segSize) * input_CHW;
        int offset2 = (i + 2 * segSize) * input_CHW;
        int offset3 = (i + 3 * segSize) * input_CHW;

        // output offset
        int out_offset0 = (i + 0 * segSize) * output_MHW;
        int out_offset1 = (i + 1 * segSize) * output_MHW;
        int out_offset2 = (i + 2 * segSize) * output_MHW;
        int out_offset3 = (i + 3 * segSize) * output_MHW;

        // async cpy input, host to device
        cudaMemcpyAsync(*device_input_ptr + offset0, host_input + offset0, in_copy_size * sizeof(float), cudaMemcpyHostToDevice, streams[0]);
        cudaMemcpyAsync(*device_input_ptr + offset1, host_input + offset1, in_copy_size * sizeof(float), cudaMemcpyHostToDevice, streams[1]);
        cudaMemcpyAsync(*device_input_ptr + offset2, host_input + offset2, in_copy_size * sizeof(float), cudaMemcpyHostToDevice, streams[2]);
        cudaMemcpyAsync(*device_input_ptr + offset3, host_input + offset3, in_copy_size * sizeof(float), cudaMemcpyHostToDevice, streams[3]);

        // stream kernel calls
        conv_forward_kernel<<<DimGrids, DimBlocks, 0, streams[0]>>>(*device_output_ptr + out_offset0, *device_input_ptr + offset0, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);
        conv_forward_kernel<<<DimGrids, DimBlocks, 0, streams[1]>>>(*device_output_ptr + out_offset1, *device_input_ptr + offset1, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);
        conv_forward_kernel<<<DimGrids, DimBlocks, 0, streams[2]>>>(*device_output_ptr + out_offset2, *device_input_ptr + offset2, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);
        conv_forward_kernel<<<DimGrids, DimBlocks, 0, streams[3]>>>(*device_output_ptr + out_offset3, *device_input_ptr + offset3, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);

        // async cpy output, device to host
        cudaMemcpyAsync((void*)(host_output + out_offset0), *device_output_ptr + out_offset0, out_copy_size * sizeof(float), cudaMemcpyDeviceToHost, streams[0]);
        cudaMemcpyAsync((void*)(host_output + out_offset1), *device_output_ptr + out_offset1, out_copy_size * sizeof(float), cudaMemcpyDeviceToHost, streams[1]);
        cudaMemcpyAsync((void*)(host_output + out_offset2), *device_output_ptr + out_offset2, out_copy_size * sizeof(float), cudaMemcpyDeviceToHost, streams[2]);
        cudaMemcpyAsync((void*)(host_output + out_offset3), *device_output_ptr + out_offset3, out_copy_size * sizeof(float), cudaMemcpyDeviceToHost, streams[3]);
    }

    cudaFree(*device_input_ptr);
    cudaFree(*device_output_ptr);
}
```

Other changed part of the code



```
__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input, con
{
    // Set the kernel dimensions and call the kernel
    // int H_out = Height - K + 1;
    // int W_out = Width - K + 1;

    // // defined the grids required for the output
    // int H_grid = ceil((H_out*1.0)/TILE_WIDTH);  // height grids
    // int W_grid = ceil((W_out*1.0)/TILE_WIDTH);  // width grids
    // int Z = H_grid * W_grid;                    // h x w requirements for the image


    return;
    // cudaDeviceSynchronize();

}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, flo
{
    // Copy the output back to host
    // int H_out = Height - K + 1;
    // int W_out = Width - K + 1;
    // Free device memory
    return;
}
```