Name: *Elijah Gaohan Ye*
NetID: *gaohany2*
Section: *AL, OLC*

ECE 408/CS483 Milestone 3 Report

All of my code is in /Project/custom/src, I labeled them by the optimization. Thanks!

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 for this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.1804 ms*	*0.631415 ms*	*0m1.227s*	*0.86*
1000	*1.56452 ms*	*11.0054 ms*	*0m9.856s*	*0.886*
10000	*16.0549 ms*	*62.5924 ms*	*1m36.681s*	*0.8714*

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

    a. Which optimization did you choose to implement? Choose from the optimization below by clicking on the check box and explain why you chose 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 to implement constant memory for weight matrix(mask) because the mask is read only, and we use it a lot. Instead of accessing it from Global Memory which

normally takes hundreds of clock cycles, accessing from Constant Memory only takes around 5 clock cycles with caching. (Lecture 4 slide).

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 moving all the mask onto Constant Memory. I think it would increase the performance of the forward convolution since accessing Constant Memory is faster than accessing Global Memory. Since this is my first optimization it does not synergize with my previous optimization. Though I think this would works for most of the 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	*0.158498 ms*	*0.578215 ms*	*0m1.223s*	*0.86*
1000	*1.47798 ms*	*5.64542 ms*	*0m9.723s*	*0.886*
10000	*14.5487 ms*	*57.4982 ms*	*1m38.712s*	*0.8714*

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

This optimization successfully decreased the op times by a little bit. It's a small improvement, less than I expected. One of the TAs mentioned on CampusWire that if the original baseline OP time is very low already, the amount of optimization that Constant Memory can bring is relatively low. (See post 326)

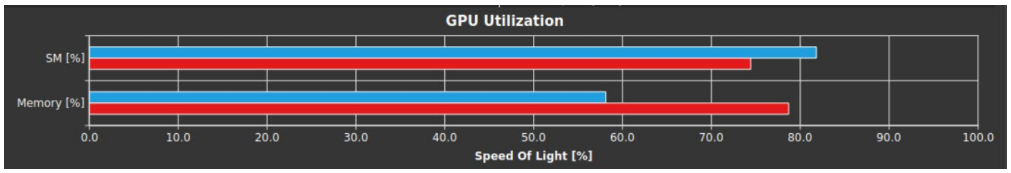

Figure 1. GPU Utilization Constant vs Baseline

The red is baseline, and blue is constant memory. We can see that through our optimization, we brought the SM usage up and brought the memory usage down.

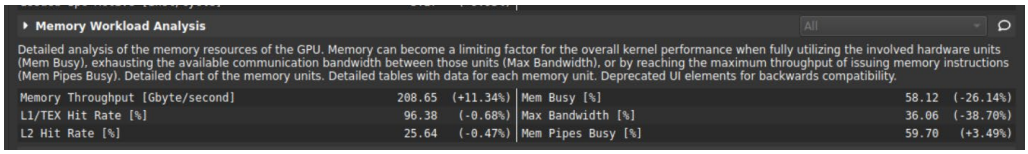

Figure 2. Memory Workload Analysis Constant vs Baseline

The baseline implementation has a very high L1/TEX Hit Rate. However, through our optimization, we still manage to achieve an 11.34% increase in our Memory Throughput which normally is the bottle neck for the performance.

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

Lecture 4 and MP4 helped me to implement this technique.

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.

```
__constant__ float device_kernel[16 * 7 * 7 * 4]; // from campuswire post #308
// Tile width here is 16 and the kernel length should be 1 * 7 * 7 * 4*16 -- TA
Tao Huili
__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;

    int W_grid = ceil(Width_out/(TILE_WIDTH * 1.0));

    #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 kernel_4d(i3, i2, i1, i0) device_kernel[(i3) * (Channel * K * K) +
(i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here
    int n, m, h, w, c, p, q;
    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 = 0.0f;

    if(h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c) {
            for(p = 0; p < K; ++p) {
                for(q = 0; q < K; ++q) {
                    // acc += in_4d(n, c, h+p, w+q) * mask_4d(m, c, p, q);
                    acc += in_4d(n, c, h+p, w+q) * kernel_4d(m, c, p, q);
                }
            }
        }
        out_4d(n,m,h,w) = acc;
    }


    #undef out_4d
    #undef in_4d
    #undef kernel_4d
}
```

2. **Optimization 2: *Tiled Shared Memory***

    a. Which optimization did you choose to implement? Choose from the optimization below by clicking on the check box and explain why you chose 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 tiled shared memory for a reason similar to last optimization. Accessing shared memory is faster than global memory. Although it takes time to load into the shared memory but once you loaded into the share memory, it can be reused. Again, read/write per-block shared memory is 5 clock cycles, and read/write per grid global memory is around 500 clock cycles. (Lecture 4 slides 9)

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?

We will have a shared memory tile of input for each block. In the beginning, we will need to load the value from global memory into shared memory. Then we will compute the output using the input in shared memory and the mask in global memory. This optimization can synergize with the previous optimization since one loads the input into shared memory and one loads the mask into constant memory. This optimization might increase the performance by very little amount. The reason is because the scope for shared memory is per block, and the scope for constant memory is per grid. So, for each block we will need to reload, which is accessing global memory and that is slow.

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	*0.187748 ms*	*0.767605 ms*	*0m1.281s*	*0.86*
1000	*1.75736 ms*	*7.52134 ms*	*0m10.163s*	*0.886*
10000	*17.3863 ms*	*74.3311 ms*	*1m36.955s*	*0.8714*

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

From its paper stats, I will say the optimization is not successful since the OP increased from baseline implementation.

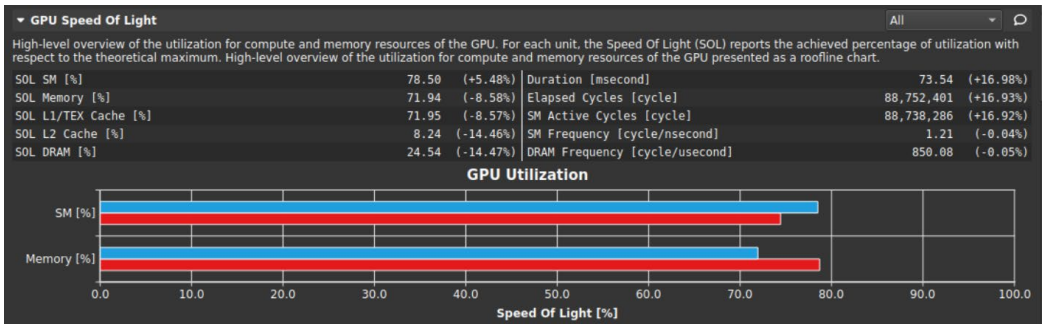

Figure 3. GPU Utilization: Shared vs Baseline

Here we can see that the increase from SM% is less than the increase constant memory when comparing with the baseline, and memory% decrease is also less than constant memory. So, it is similar to our prediction that it might not increase the performance.

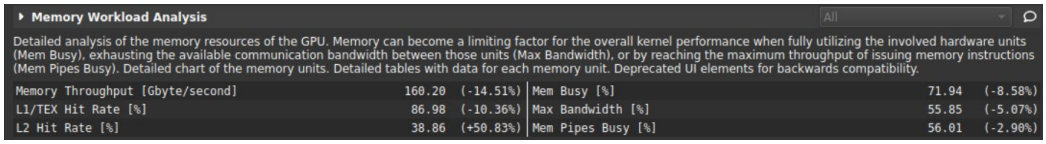

Figure 4: Memory Workload Analysis: Shared vs Baseline

From Figure 4, we can also see the L1/TEX Hit Rate dropped 10%. Though L2 Hit Rate increased by 50% due to Shared Memory, the total Memory Throughput decreased by 14.5% compared to baseline. I think this is the main cause for the performance decrease.

Kernel Name	*Time %*	*Total time (ns)*	*Name*
Shared	*100*	*90207673*	*conv_forward_kernel*
Baseline	*100*	*79450540*	*conv_forward_kernel*

Figure 5: Nsys: Shared vs Baseline

Even from Nsys, we see that most of the increase is from CUDA Kernel Statistics.

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

I used lecture 8 and MP4 to implement this technique.

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.

```
__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;

    int W_grid = ceil(Width_out/(TILE_WIDTH * 1.0));
    int sm_width = TILE_WIDTH + K - 1;

    extern __shared__ float tile[];

    #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) mask[(i3) * (Channel * K * K) + (i2) * (K *
K) + (i1) * (K) + i0]
    #define tile_3d(i2, i1, i0) tile[(i2) * (sm_width * sm_width) + (i1) *
(sm_width) + i0]

    // Insert your GPU convolution kernel code here
    int n, m, h, h0, w, w0, c, p, q, i, j;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    n = blockIdx.x;
    m = blockIdx.y;
    h0 = (blockIdx.z / W_grid) * TILE_WIDTH;
    h = h0 + ty;
    w0 = (blockIdx.z % W_grid) * TILE_WIDTH;
    w = w0 + tx;

    // load data into shared memory
    for(c = 0; c < Channel; ++c){
        for(i = ty; i < sm_width; i += TILE_WIDTH){
            for(j = tx; j < sm_width; j += TILE_WIDTH){
                if(h0 + i < Height && w0 + j < Width){
                    tile_3d(c, i, j) = in_4d(n, c, h0 + i, w0 + j);
                }
            }
        }
    }

    __syncthreads();
```

```
    if(h < Height_out && w < Width_out){
        float acc = 0.0f;
        for(c = 0; c < Channel; ++c) {
            for(p = 0; p < K; ++p) {
                for(q = 0; q < K; ++q) {
                    // acc += in_4d(n, c, h+p, w+q) * mask_4d(m, c, p, q);
                    acc += tile_3d(c, p + ty, q + tx)* mask_4d(m, c, p, q);
                }
            }
        }
        out_4d(n,m,h,w) = acc;
    }


    #undef out_4d
    #undef in_4d
    #undef mask_4d
    #undef tile_3d
}
```

In my kernel function call, there is one thing that is very important which sets the size of shared memory through kernel call.

```
conv_forward_kernel<<<gridDim, blockDim,
numSharedElements*sizeof(float)>>>(device_output, device_input, device_mask,
Batch, Map_out, Channel, Height, Width, K);
```

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

   a. Which optimization did you choose to implement? Choose from the optimization below by clicking on the check box and explain why you chose 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 fixed point arithmetic because fixed-point arithmetic operations can be faster than floating-point operations in CUDA. This is because fixed-point arithmetic requires less memory bandwidth and computation resources than floating-point arithmetic.

   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 mainly works by using a library called cuda_fp16. (CUDA Math API :: CUDA Toolkit Documentation (nvidia.com)) In CUDA, the fp 16 is called half for the data type. We used some functions such as __float2half (converting a float type to fp 16), __hadd (adding two half (fp 16)), __hmul(multiplying two half (fp 16)), __half2float (converting a fp 16 to a normal float). Depending how you implement it. I think there are two ways to implement this: one is you do everything in one kernel, the other one is you have separate kernels for different tasks.

I did everything in one so that one kernel actually takes a longer time. Since I convert to half, then do the multiplication and add, and then convert it back into float in one kernel. This is slower than just doing normal float operations. However, I believe, if you prepared the data in half already and perform everything in half. This is faster than float. However, the overall performance might still take longer than normal float since you need to prepare the data in the beginning and convert the data back in the end. It can synergize with all the previous optimizations. However, it will not help with the overall performance due to the reason I mentioned above.

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	*0.22511 ms*	*0.833481 ms*	*0m1.237s*	*0.86*
1000	*2.09651 ms*	*8.16526 ms*	*0m9.699s*	*0.887*
10000	*20.5938 ms*	*80.4963 ms*	*1m34.991s*	*0.8716*

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

This implementation is both successful and unsuccessful in improving performance.

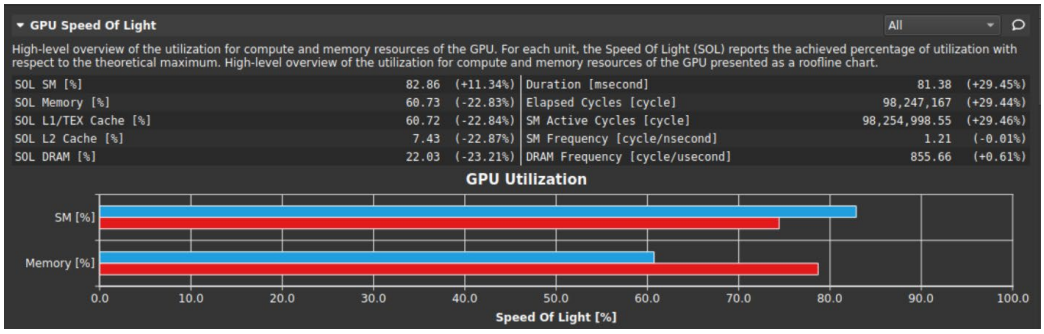

Figure 6. GPU Utilization: FP 16 vs Baseline

Success, we see an increase in SM% and decrease in Memory %. The decrease in Memory % is probably achieved through float to fp 16 conversion, since fp 16 uses less memory due to the bits it needs is only 16bit.

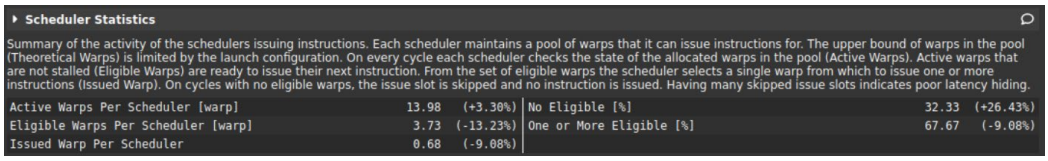

Figure 7. Scheduler Statistics: FP 16 vs Baseline

From Figure 6, we see that Eligible Warps Per Scheduler decreased by 13.2%. This is primarily due to the overhead of preparing the data inside one kernel. So, the overall time for this kernel increased.

Kernel Name	*Time %*	*Total time (ns)*	*Name*
FP 16	*100*	*102285497*	*conv_forward_kernel*
Baseline	*100*	*79450540*	*conv_forward_kernel*

Figure 8: Nsys: FP 16 vs Baseline

From Figure 7, we see a pretty significant increase of total kernel time. Roughly about 20%. Hence, it is both successful and unsuccessful.

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

I mainly use the source from online. [CUDA Math API :: CUDA Toolkit Documentation (nvidia.com)](). This link helped me a lot during the process.

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.

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

#define TILE_WIDTH 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;

    int W_grid = ceil(Width_out/(TILE_WIDTH * 1.0));
```

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

    // Insert your GPU convolution kernel code here
    int n, m, h, w, c, p, q;
    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 = 0.0;

    if(h < Height_out && w < Width_out){
        for(c = 0; c < Channel; ++c) {
            for(p = 0; p < K; ++p) {
                for(q = 0; q < K; ++q) {
                    acc = __hadd(acc, __hmul(__float2half(in_4d(n, c, h+p,
w+q)), __float2half(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***

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

   So, during lecture 22, we talked about CUDA supports parallel execution of kernels and cudaMemcpy with streams. Basically, each stream is a queue of operations, and tasks in different streams can be executed in parallel. In this way, we can increase the overall 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 using cudaStream. First you need to create several streams. You need to define them first by cudaStream_t and then using cudaStreamCreate which takes the ptr/addr to a stream. Then depending on the SegSize you defined, you will put these streams in a for loop and try to run them until the batch size in our project. In the lecture 22 slides 15, it also talked about how to align the tasks in the for loop in order to overlap as much as we can.

   I think it will increase the performance of the forward convolution since we are dividing the tasks and execute them parallelly. This can synergize with our constant memory optimization since mask is of different size than input and output and it is not depends on the batch size, so we did not stream the loading of our mask, and we still access it from the global memory. So, it would be nice if we combined these two and we should see a 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	*0.005145 ms*	*0.004793 ms*	*0m1.194s*	*0.86*
1000	*0.005783 ms*	*0.007722 ms*	*0m9.920s*	*0.886*
10000	*0.006255 ms*	*0.005963 ms*	*1m37.951s*	*0.8714*

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

This optimization gives you an illusion of improving performance since the op times are so fast. This is because we run them in parallel. But if we check the real time and Nsys's CUDA Kernel Statistics, we can see it actually might not improve like we thought.

First, let's take a look at Nsight Compute.

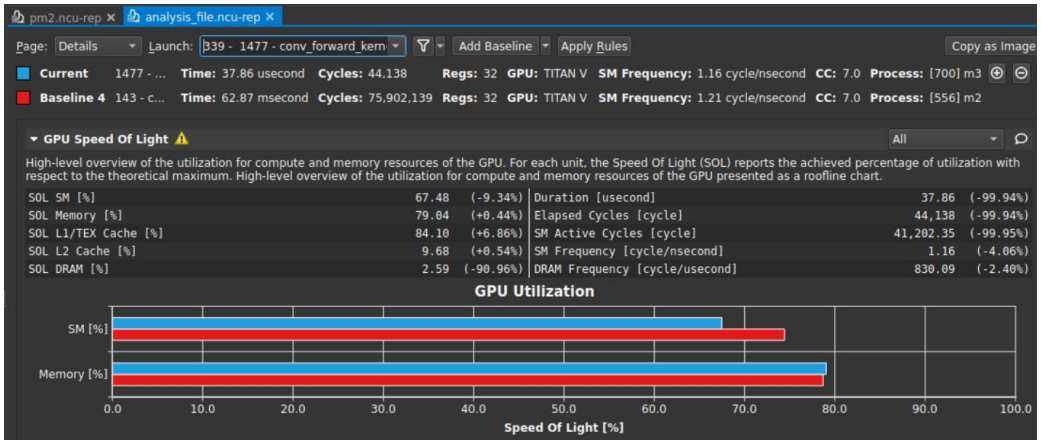

Figure 8. GPU Utilization: Stream vs Baseline

We can see that the GPU Utilization actually did not change much. One of the reasons is because we did not modify the kernel code, instead, we fired tons of kernels in parallel and gathered all the results. At top left, you will also see that the baseline kernel runs for 62.87 ms whereas the stream kernel runs for only 37.86 microseconds. However, there are hundreds of these small kernels in our stream.

Kernel Name	*Time %*	*Total time (ns)*	*Instances*	*Name*
Streams	*100*	*98851625*	*1000*	*conv_forward_kernel*
Baseline	*100*	*79450540*	*2*	*conv_forward_kernel*

Figure 9. Nsys CUDA Kernel Statistics

Nsys also tells us the same story that Stream kind of "cheated" its way to look a lot better in first glance. However, when we take a deeper look, it might not be all that pretty as we thought.

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

Mostly 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.

I did not change the kernel from the Baseline implementation so I will not paste my kernel. However, I changed a lot from the host code.

Here's the host code snippet.

```
__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 Batch, const int
Map_out, const int Channel, const int Height, const int Width, const int K)
{
    // Allocate memory and copy over the relevant data structures to the GPU

    /* Input: 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 Batch, const int Map_out, const int Channel, const int
Height, const int Width,
     *  const int K */
    unsigned int numInputElements;
    unsigned int numOutputElements;
    unsigned int numMaskElements;

    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;

    numInputElements = Batch * Channel * Height * Width;
    numOutputElements = Batch * Map_out * Height_out * Width_out;
    numMaskElements = Map_out * Channel * K * K;

    cudaMalloc((void **) device_input_ptr, numInputElements * sizeof(float));
    cudaMalloc((void **) device_output_ptr, numOutputElements * sizeof(float));
    cudaMalloc((void **) device_mask_ptr, numMaskElements * sizeof(float));

    cudaMemcpy(*device_mask_ptr, host_mask, numMaskElements * sizeof(float),
cudaMemcpyHostToDevice);

    const int SegSize = 20;

    int W_grid = ceil(Width_out /(TILE_WIDTH * 1.0));
    int H_grid = ceil(Height_out /(TILE_WIDTH * 1.0));

    int Z = W_grid * H_grid;
    int SegSize_Input = Channel * Height * Width;
    int SegSize_Output = Map_out * Height_out * Width_out;

    dim3 gridDim(SegSize, Map_out, Z);
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);

    cudaStream_t stream0, stream1, stream2, stream3, stream4;

    cudaStreamCreate(&stream0);
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);
    cudaStreamCreate(&stream3);
    cudaStreamCreate(&stream4);

    /* Input: 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 Batch, const int Map_out, const int Channel, const int
Height, const int Width,
     *  const int K */
    for(int i = 0; i < Batch; i += 5 * SegSize){
        int stream0_input_off = (i + 0 * SegSize) * SegSize_Input;
        int stream1_input_off = (i + 1 * SegSize) * SegSize_Input;
        int stream2_input_off = (i + 2 * SegSize) * SegSize_Input;
        int stream3_input_off = (i + 3 * SegSize) * SegSize_Input;
        int stream4_input_off = (i + 4 * SegSize) * SegSize_Input;

        int stream0_output_off = (i + 0 * SegSize) * SegSize_Output;
        int stream1_output_off = (i + 1 * SegSize) * SegSize_Output;
        int stream2_output_off = (i + 2 * SegSize) * SegSize_Output;
        int stream3_output_off = (i + 3 * SegSize) * SegSize_Output;
        int stream4_output_off = (i + 4 * SegSize) * SegSize_Output;



        cudaMemcpyAsync(*device_input_ptr + stream0_input_off, host_input +
stream0_input_off, SegSize*SegSize_Input*sizeof(float), cudaMemcpyHostToDevice,
stream0);
        cudaMemcpyAsync(*device_input_ptr + stream1_input_off, host_input +
stream1_input_off, SegSize*SegSize_Input*sizeof(float), cudaMemcpyHostToDevice,
stream1);
        cudaMemcpyAsync(*device_input_ptr + stream2_input_off, host_input +
stream2_input_off, SegSize*SegSize_Input*sizeof(float), cudaMemcpyHostToDevice,
stream2);
        cudaMemcpyAsync(*device_input_ptr + stream3_input_off, host_input +
stream3_input_off, SegSize*SegSize_Input*sizeof(float), cudaMemcpyHostToDevice,
stream3);
        cudaMemcpyAsync(*device_input_ptr + stream4_input_off, host_input +
stream4_input_off, SegSize*SegSize_Input*sizeof(float), cudaMemcpyHostToDevice,
stream4);



        conv_forward_kernel<<<gridDim, blockDim, 0,
stream0>>>(*device_output_ptr + stream0_output_off, *device_input_ptr +
stream0_input_off, *device_mask_ptr, Batch, Map_out, Channel, Height, Width,
K);
        conv_forward_kernel<<<gridDim, blockDim, 0,
stream1>>>(*device_output_ptr + stream1_output_off, *device_input_ptr +
stream1_input_off, *device_mask_ptr, Batch, Map_out, Channel, Height, Width,
K);
        conv_forward_kernel<<<gridDim, blockDim, 0,
stream2>>>(*device_output_ptr + stream2_output_off, *device_input_ptr +
```

```
stream2_input_off, *device_mask_ptr, Batch, Map_out, Channel, Height, Width,
K);
        conv_forward_kernel<<<gridDim, blockDim, 0,
stream3>>>(*device_output_ptr + stream3_output_off, *device_input_ptr +
stream3_input_off, *device_mask_ptr, Batch, Map_out, Channel, Height, Width,
K);
        conv_forward_kernel<<<gridDim, blockDim, 0,
stream4>>>(*device_output_ptr + stream4_output_off, *device_input_ptr +
stream4_input_off, *device_mask_ptr, Batch, Map_out, Channel, Height, Width,
K);


        cudaMemcpyAsync((void *)(host_output + stream0_output_off),
*device_output_ptr + stream0_output_off, SegSize*SegSize_Output*sizeof(float),
cudaMemcpyDeviceToHost, stream0);
        cudaMemcpyAsync((void *)(host_output + stream1_output_off),
*device_output_ptr + stream1_output_off, SegSize*SegSize_Output*sizeof(float),
cudaMemcpyDeviceToHost, stream1);
        cudaMemcpyAsync((void *)(host_output + stream2_output_off),
*device_output_ptr + stream2_output_off, SegSize*SegSize_Output*sizeof(float),
cudaMemcpyDeviceToHost, stream2);
        cudaMemcpyAsync((void *)(host_output + stream3_output_off),
*device_output_ptr + stream3_output_off, SegSize*SegSize_Output*sizeof(float),
cudaMemcpyDeviceToHost, stream3);
        cudaMemcpyAsync((void *)(host_output + stream4_output_off),
*device_output_ptr + stream4_output_off, SegSize*SegSize_Output*sizeof(float),
cudaMemcpyDeviceToHost, stream4);
    }


}
```