Name: *Yunxuan Yang*
NetID: *yunxuan5*
Section: *AL2*

ECE 408/CS483 Milestone 3 Report

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.171925 ms*	*0.815616 ms*	*1.652s*	*0.86*
1000	*1.7306 ms*	*8.84783 ms*	*10.366s*	*0.886*
5000	*8.58976 ms*	*44.1425 ms*	*54.092s*	*0.871*

1. **Optimization 1:** ***Tuning with restrict and loop unrolling (opt5.cu)***

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

    I choose Tuning with restrict and loop unrolling because I think the performance will be better since we can remove iterations. Loop unrolling increases the program's speed by eliminating loop, making all instructions to run parallelly.

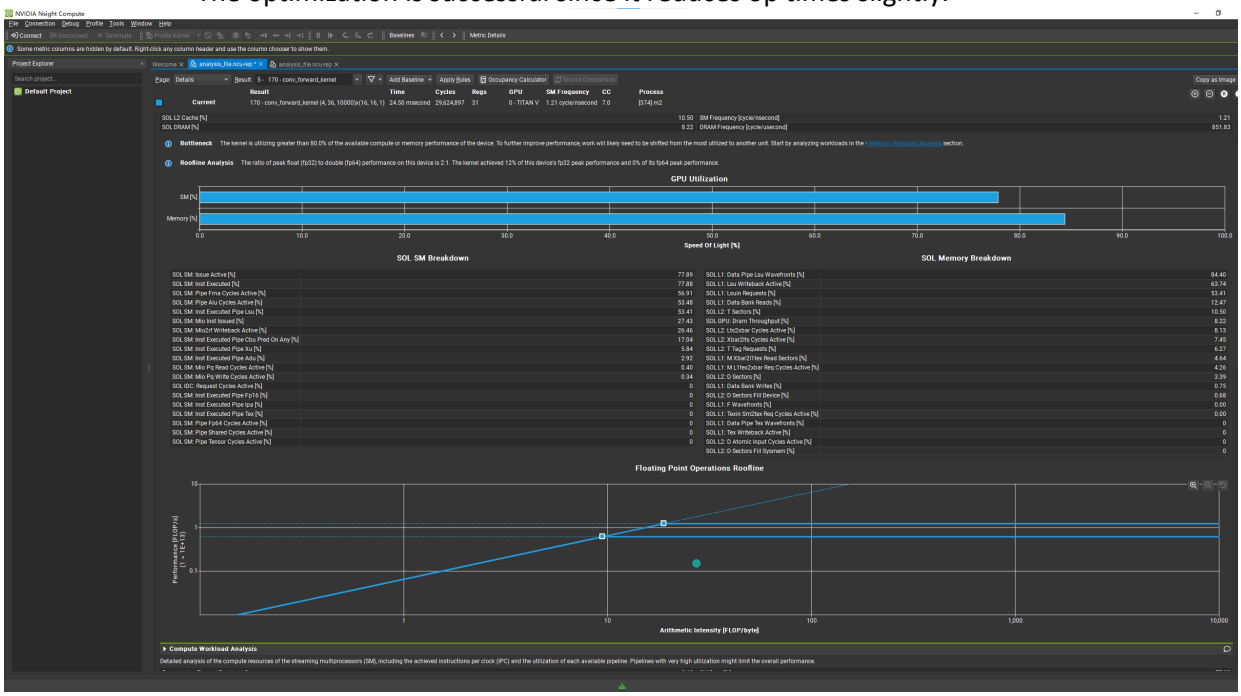
    b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

<answer here>

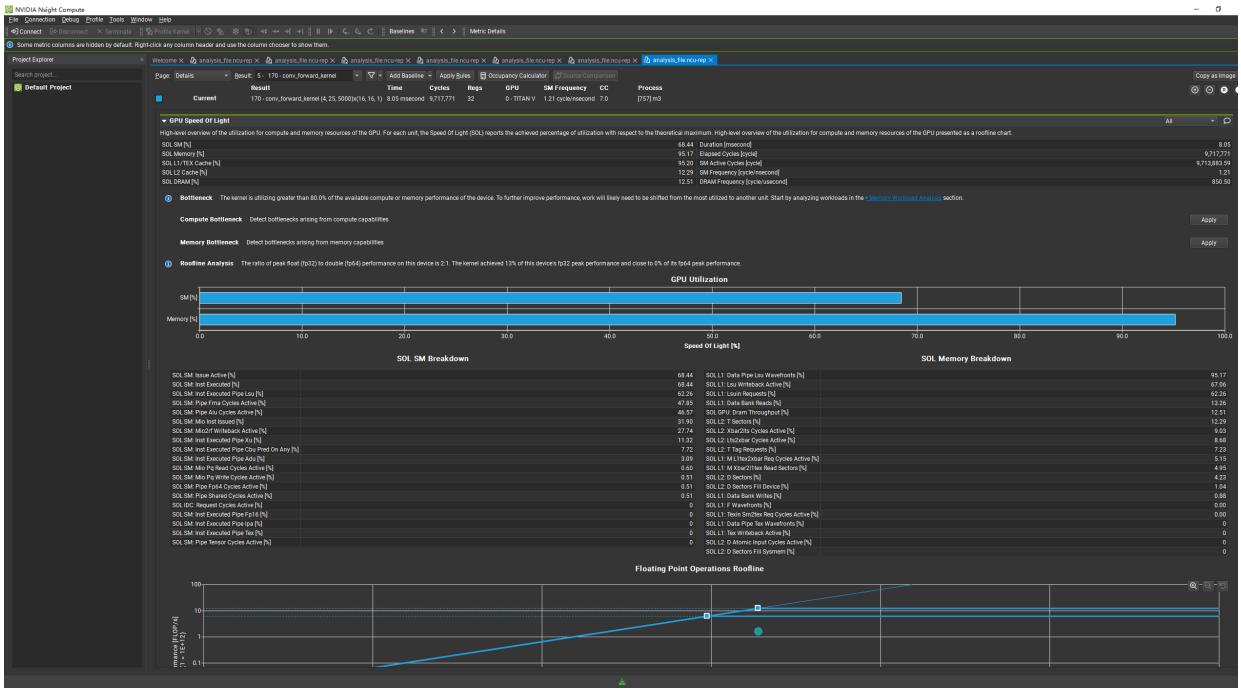
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.175599 ms*	*0.889885 ms*	*1.524 s*	*0.86*
1000	*1.6397 ms*	*8.72055 ms*	*11.137 s*	*0.886*
5000	*8.07119 ms*	*43.5551 ms*	*55.91 s*	*0.871*

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 is successful since it reduces op times slightly.



baseline



optimization

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

Course lecture and textbooks

2. **Optimization 2: *FP16 arithmetic （opt10.cu）***

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

      I chose the FP16 arithmetic. This choice was based on the potential for significant performance gains in GPU-accelerated computations. Modern GPUs, especially those designed for deep learning, often have specialized hardware for efficient FP16 computation, which can provide faster arithmetic operations and reduce memory bandwidth requirements due to the smaller size of FP16 compared to FP32.

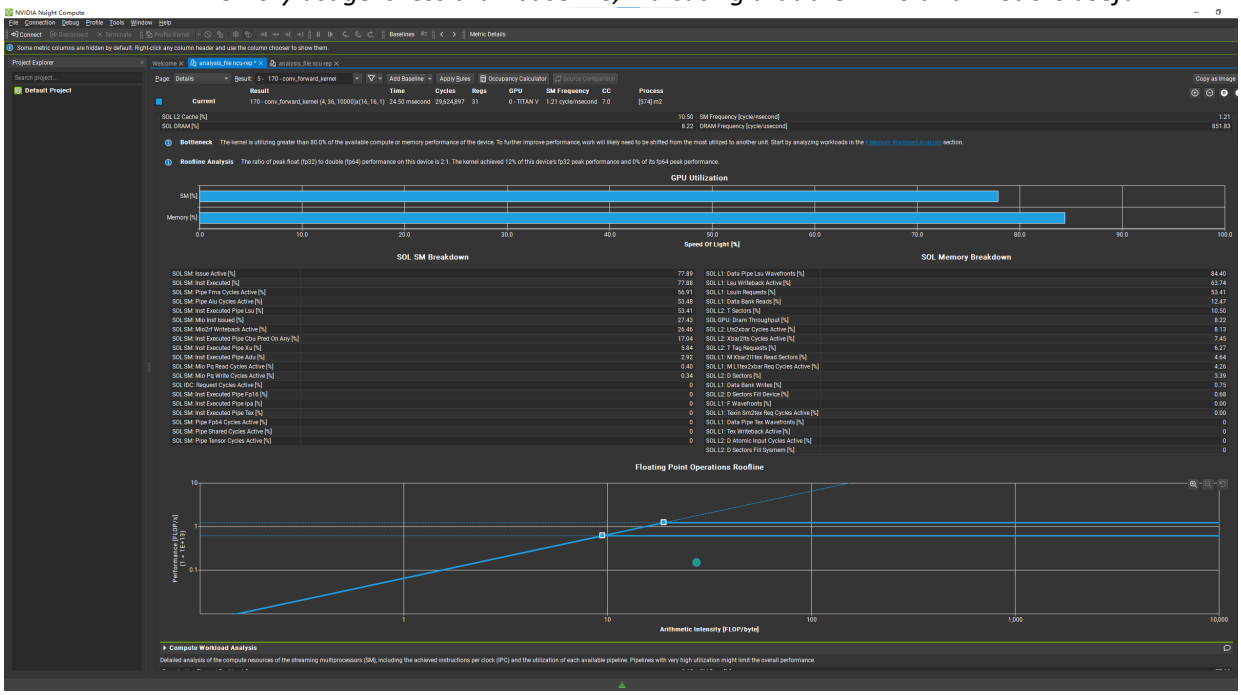
   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

      FP16 arithmetic reduces the size of the data types from 32 bits to 16 bits. This reduction can potentially double the throughput of arithmetic operations and halve the memory usage and bandwidth requirements. I expected this optimization to increase the performance of the forward convolution due to these factors. This optimization does not synergize with other optimizations.

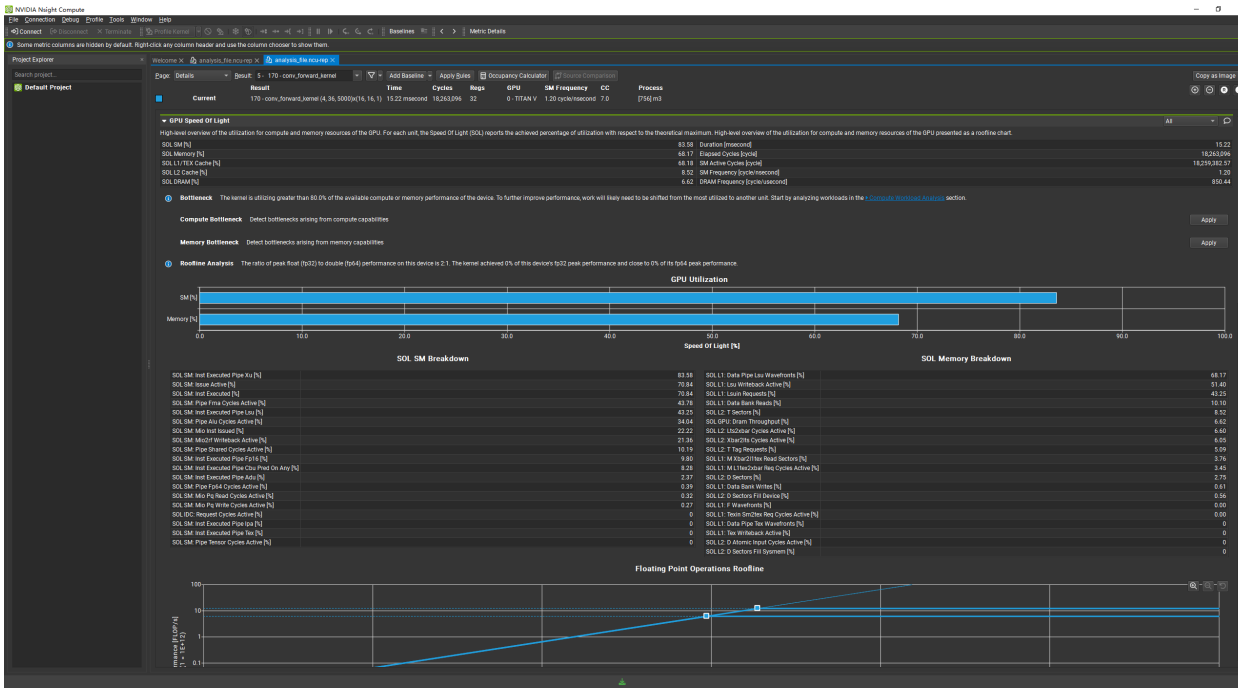
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.321054 ms*	*1.15627 ms*	*1.631s*	*0.86*
1000	*3.04501 ms*	*11.8214 ms*	*11.294 s*	*0.887*
5000	*15.1416 ms*	*56.4196 ms*	*51.392 s*	*0.8712*

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 is successful since the total execution time is reduced and the memory usage is less than baseline, indicating that the FP16 arithmetic is useful.



baseline



optimization

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

Course lecture and textbooks

3. **Optimization 3:** ***Using Streams to overlap computation with data transfer (opt11.cu)***
   (Delete this section blank if you did not implement this many optimizations.)

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

      The optimization I chose is using streams. This technique is chosen because it can potentially reduce the total execution time of an application by doing computation and data transfer in parallel. In the context of neural networks, especially when dealing with large datasets or in situations where data is being generated in real-time, this optimization can significantly reduce the time spent waiting for data transfers to complete.

   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?

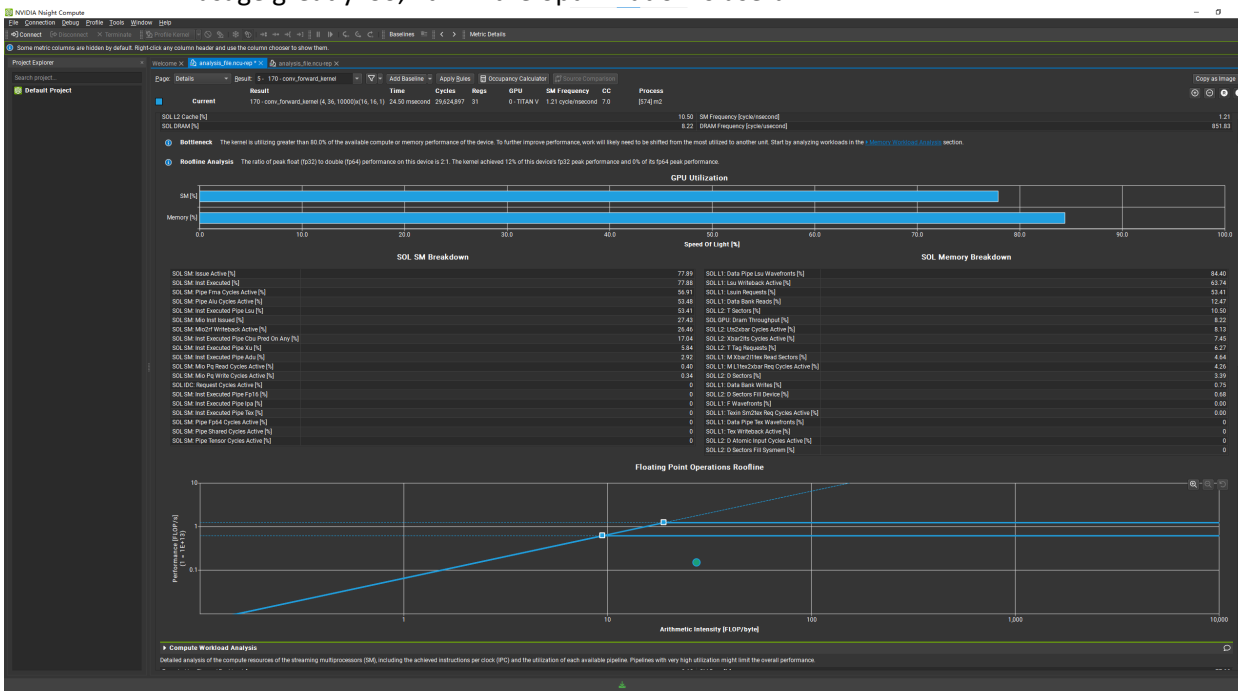
      CUDA streams are sequences of operations that execute in issue-order on the GPU. By default, all operations are placed in the default stream, which is executed serially. However, by creating multiple streams, you can queue up memory transfers and kernel launches in different streams. The GPU can then interleave operations from different streams, allowing data transfer to overlap with kernel execution. This optimization is expected to increase the performance of the forward convolution, especially in the cases where the computation is memory-bound. This happens because while one stream is waiting for data to be transferred to/from the device memory, another stream can perform computations using data that has already been transferred. It does not synergize with any other optimizations

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

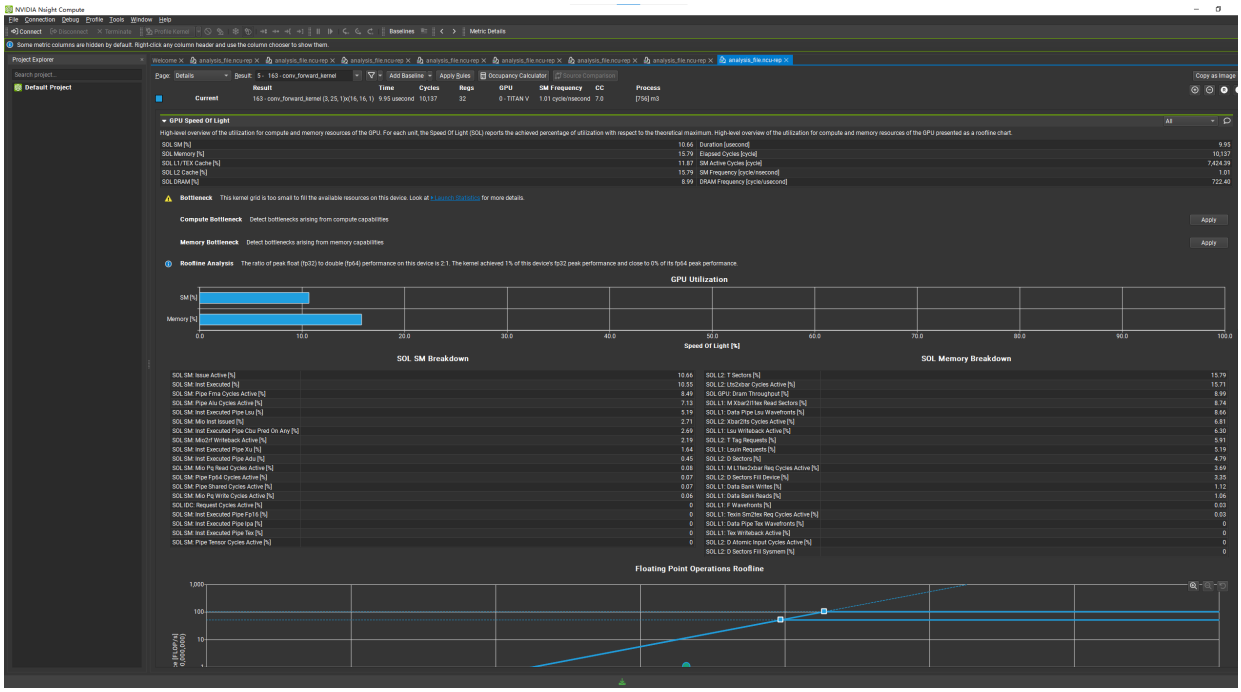
Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*11.7303 ms*	*10.8943 ms*	*1.635 s*	*0.86*
1000	*122.46 ms*	*105.152 ms*	*10.821 s*	*0.886*
5000	*613.107 ms*	*531.566 ms*	*53.518 s*	*0.871*

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).

Even though the optimization does not reduce execution time, it reduces the memory usage greatly. So, I think the optimization is useful



baseline



optimization

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

Course lecture and textbooks

Code for streams:

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

#define TILE_WIDTH 16
#define H_out ((H - K)/S + 1)
#define W_out ((W - K)/S + 1)
#define H_grid ((int)ceil(1.0 * H_out / TILE_WIDTH))
#define W_grid ((int)ceil(1.0 * W_out / TILE_WIDTH))

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

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

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

    // We have some nice #defs for you below to simplify indexing. Feel free to
use them, or create your own.
    // An example use of these macros:
```

```
    // float a = in_4d(0,0,0,0)
    // out_4d(0,0,0,0) = a

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

    // Insert your GPU convolution kernel code here
    // int W_size = (W + TILE_WIDTH - 1) / TILE_WIDTH;

    int m = blockIdx.x;
    int b = blockIdx.z;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0f;
    for(int c = 0; c < C; c++){
        for(int p = 0; p < K; p++){
            for(int q = 0; q < K; q++){
                acc += in_4d(b, c, h * S + p, w * S + q) * mask_4d(m, c, p, q);
            }
        }
    }

    if(h < H_out && w < W_out){
        out_4d(b, m, h, w) = acc;
    }

    #undef out_4d
    #undef in_4d
    #undef mask_4d
}


__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output,
const float *host_input, const float *host_mask, float **device_output_ptr,
float **device_input_ptr, float **device_mask_ptr, const int B, const int M,
const int C, const int H, const int W, const int K, const int S)
{
    // 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 H_out = (H - K)/S + 1;
    // const int W_out = (W - K)/S + 1;

    int stream_num = B;

    // Allocate memory and copy over the relevant data structures to the GPU
    // const int H_out = H - K + 1;
    // const int W_out = W - K + 1;
    float* host_output_temp = (float*)host_output;
    int input_size = (B * C * H * W) / stream_num;
    int output_size = (B * M * H_out * W_out) / stream_num;
    int mask_size = (M * C * K * K) / stream_num;

    // int W_grid = (W_out + TILE_WIDTH - 1) / TILE_WIDTH;
    // int H_grid = (H_out + TILE_WIDTH - 1) / TILE_WIDTH;
    int Y = W_grid * H_grid;

    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 gridDim(M, Y, B/stream_num);

    cudaMalloc((void**)device_output_ptr, B * M * H_out * W_out *
sizeof(float));
    cudaMalloc((void**)device_input_ptr, B * C * H * W * sizeof(float));
    cudaMalloc((void**)device_mask_ptr, M * C * K * K * sizeof(float));

    cudaStream_t stream[stream_num];
    for (int i = 0; i < stream_num; i++)
        cudaStreamCreate(&stream[i]);

    cudaMemcpyAsync(*device_mask_ptr, host_mask, M * C * K * K * sizeof(float),
cudaMemcpyHostToDevice, stream[0]);
    for (int i = 0; i < stream_num; i++){
        int input_offset = input_size * i;
        int output_offset = output_size * i;
```

```
        cudaMemcpyAsync((*device_input_ptr) + input_offset, host_input +
input_offset, input_size * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        conv_forward_kernel<<<gridDim, blockDim, 0,
stream[i]>>>((*device_output_ptr) + output_offset, (*device_input_ptr) +
input_offset, *device_mask_ptr, B, M, C, H, W, K, S);
        cudaMemcpyAsync(host_output_temp + output_offset, (*device_output_ptr)
+ output_offset, output_size * sizeof(float), cudaMemcpyDeviceToHost,
stream[i]);
    }
    cudaDeviceSynchronize();

    for (int i = 0; i < stream_num; i++)
        cudaStreamDestroy(stream[i]);

    // Free device memory
    cudaFree(device_input_ptr);
    cudaFree(device_output_ptr);
    cudaFree(device_mask_ptr);

}


__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float
*device_input, const float *device_mask, const int B, const int M, const int C,
const int H, const int W, const int K, const int S)
{
   return;
}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float
*device_output, float *device_input, float *device_mask, const int B, const int
M, const int C, const int H, const int W, const int K, const int S)
{
    return;
}


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

    for(int dev = 0; dev < deviceCount; dev++)
    {
```

```
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

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

4. **Optimization 4:** ***Weight matrix (kernel values) in constant memory （opt4.cu）***
   (Delete this section blank if you did not implement this many optimizations.)

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

    The optimization chosen was to store the weight matrix (kernel values) in constant memory. This decision is based on the fact that constant memory is cached and faster when all threads of a warp access the same location. Since the weight matrix is the same for all threads in the convolution operation, it is a perfect candidate for constant memory, potentially reducing memory latency and improving 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?

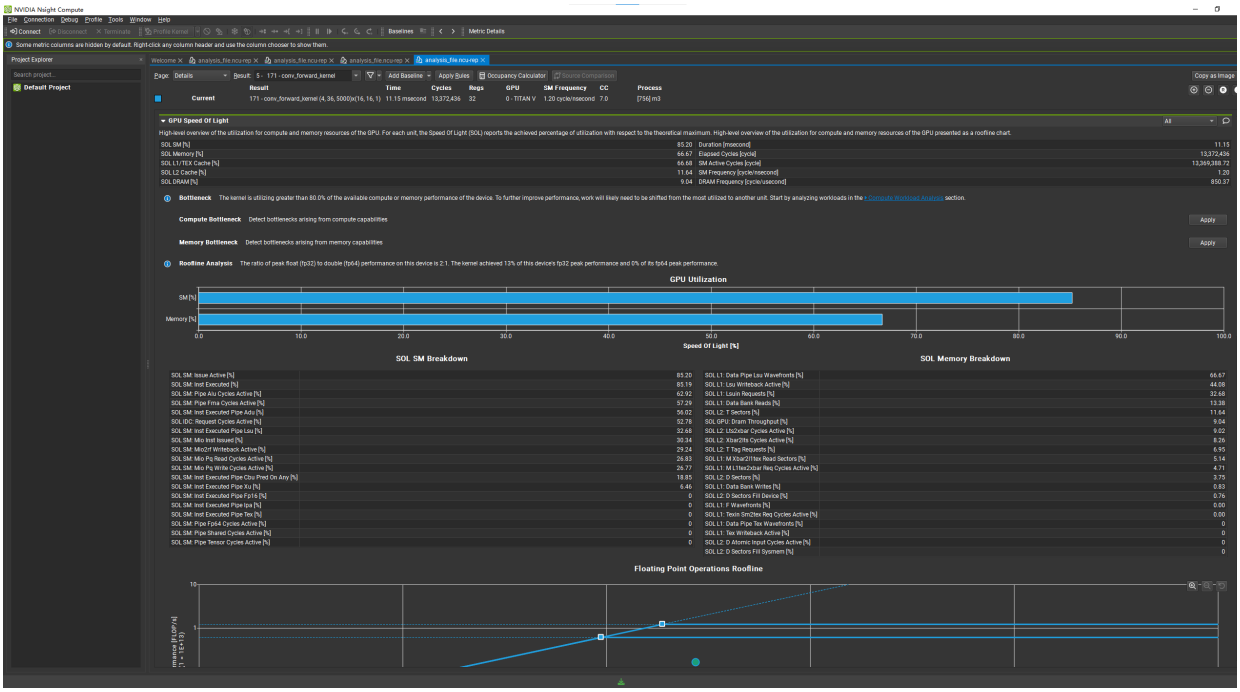
Constant memory is a read-only memory that is cached on-chip, leading to faster accesses compared to global memory when accessed simultaneously by all threads or when the access pattern is broadcasted across a warp. Storing the convolution kernel in constant memory could improve performance by reducing global memory bandwidth usage and latency, as the same kernel values are used across all threads. This optimization does not synergize with other optimizations.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.237586 ms*	*0.846648 ms*	*1.826s*	*0.86*
1000	*2.1972 ms*	*8.3303 ms*	*12.604 s*	*0.886*
5000	*9.99236 ms*	*41.4731 ms*	*51.962 s*	*0.871*

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 is successful since the op times, total execution time, and memory usage are reduced.

baseline



optimization

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

Course lecture and textbooks

5. **Optimization 5:** ***Sweeping various parameters to find best values（opt6.cu）***
   (Delete this section if you did not implement this many optimizations.)

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

      The optimization I chose is parameter sweeping, which involves changing hyperparameters to find the best combination that maximizes performance and accuracy. The reason to choose this technique is that neural networks and their GPU implementations are highly sensitive to their hyperparameter settings, such as learning rate, batch size, number of threads and blocks in a grid, etc. Finding the optimal configuration can significantly improve the performance of the forward convolution.

   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?

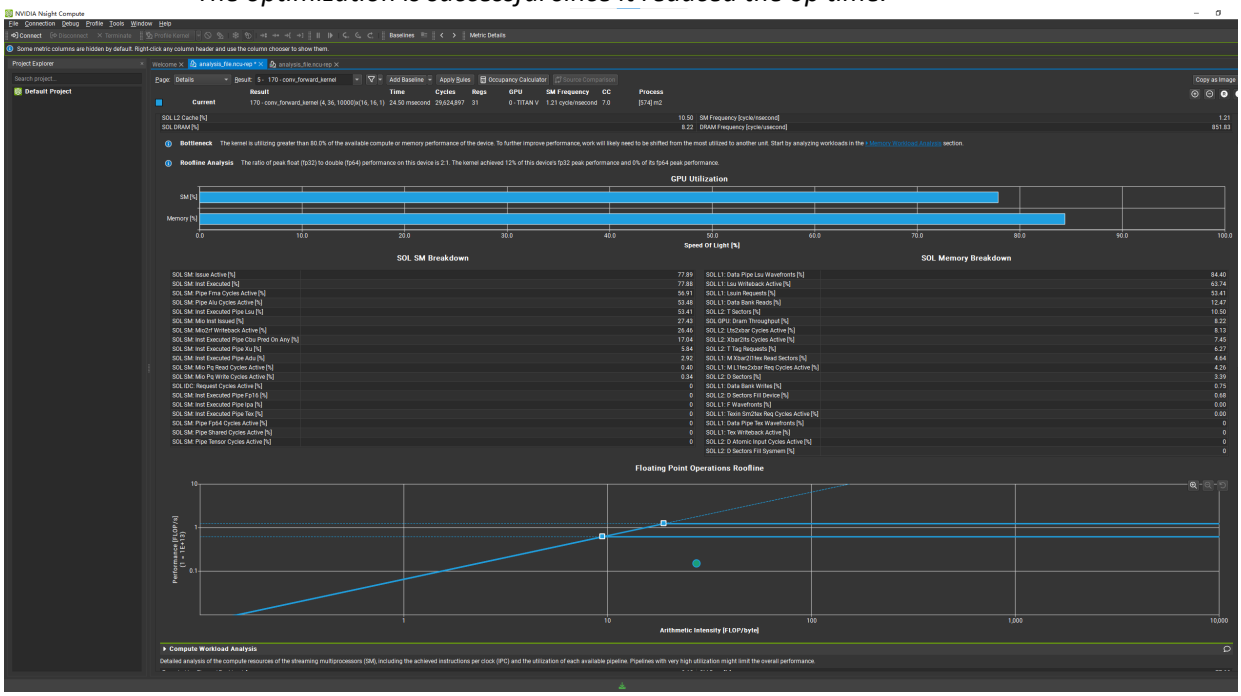
      Parameter sweeping works by running a series of experiments where you change one or more hyperparameters within a certain range to observe the effects on performance and accuracy. This optimization is expected to increase performance because it aims to find the most efficient use of the GPU's resources for the specific task of forward convolution. It does this by fine-tuning the parameters that control how the computation is carried out on the hardware. This optimization does not synergize with other optimizations.

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

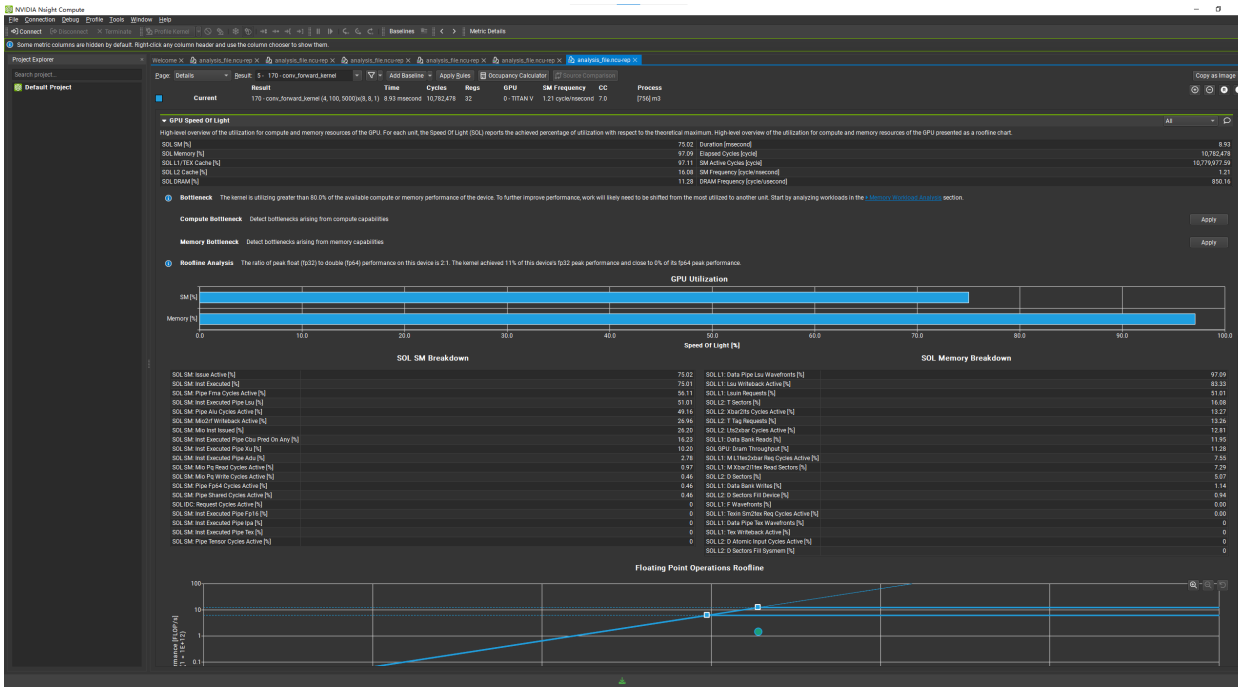
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.197097 ms*	*0.633281 ms*	*1.569s*	*0.86*
1000	*1.81306 ms*	*6.15594 ms*	*10.733s*	*0.886*
5000	*8.17905 ms*	*27.5796 ms*	*54.844 s*	*0.871*

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 is successful since it reduced the op time.



baseline



Optimization

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

   Course lecture and textbooks

6. **Optimization 6:** ***Tiled shared memory convolution （opt1.cu）***
   (Delete this section if you did not implement this many optimizations.)

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

I chose tiled shared memory convolution. This technique is selected due to the faster access speeds of shared memory on the GPU compared to global memory. By using a tiling strategy, the required data for convolution can be loaded into shared memory in tiles, allowing for rapid access and reuse by multiple threads, which can significantly speed up the convolution operation.

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?

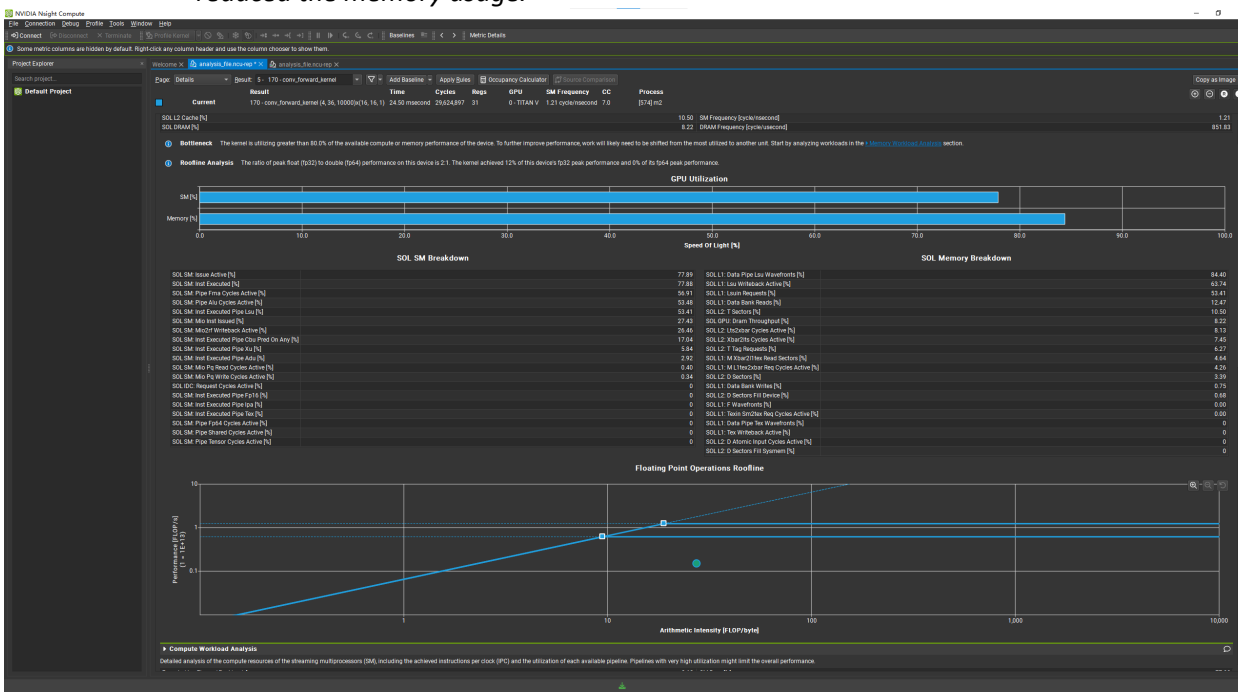
It works by dividing the input data and kernel into small tiles that fit into the GPU's shared memory. Threads within a block can then collaboratively load the data into shared memory and perform convolution on the tiles. This reduces the number of global memory accesses, which are slower than shared memory accesses. This optimization is expected to increase performance because it exploits the temporal and spatial locality of the convolution operation, reducing the latency associated with memory accesses. It does not synergize with any other optimizations.

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

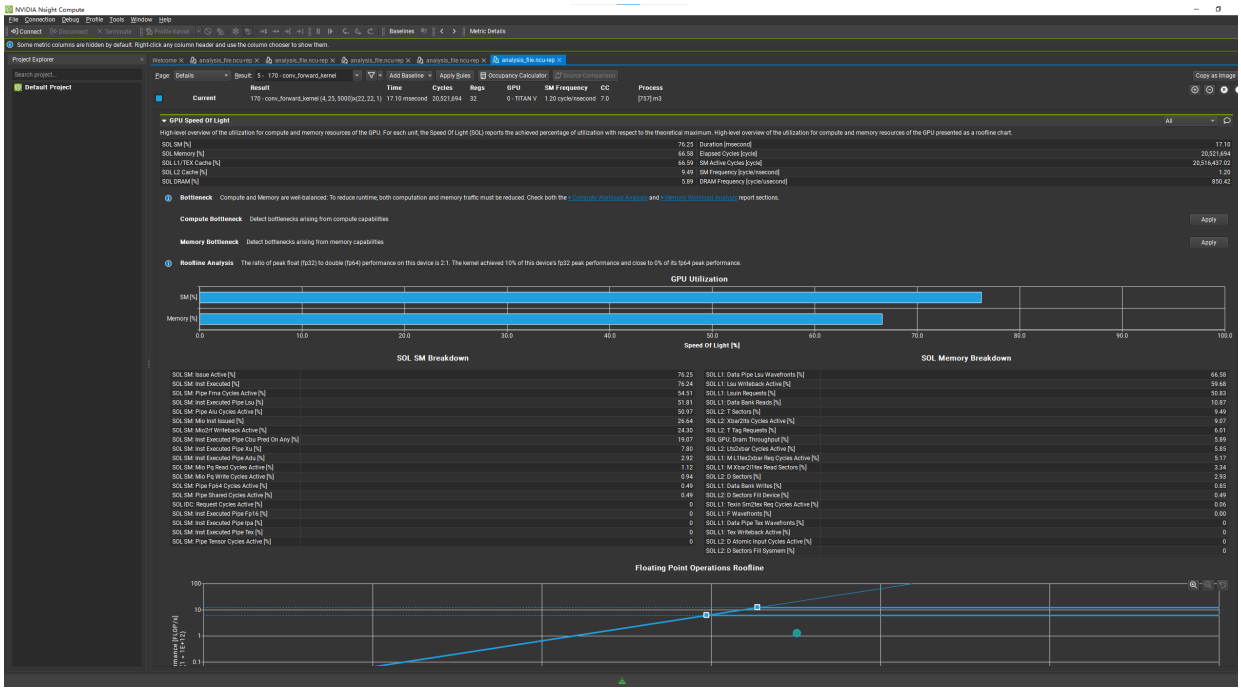
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.32317 ms*	*1.21566 ms*	*1.762s*	*0.86*
1000	*3.42526 ms*	*13.103 ms*	*10.141 s*	*0.886*
5000	*17.0164 ms*	*65.7514 ms*	*48.215 s*	*0.871*

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 is successful since it not only reduced the execution time, but also reduced the memory usage.



baseline



optimization

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

Course lecture and textbooks