Name:	*Rishabh Garg*
NetID:	*rg18*
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	*0.196133 ms*	*0.612998 ms*	*0m1.256s*	*0.86*
1000	*1.82594 ms*	*5.88128 ms*	*0m10.030s*	*0.886*
10000	*17.9873 ms*	*58.9158 ms*	*1m39.434s*	*0.8714*

1. **Optimization 1: Tiled shared memory convolution**

   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

Convolutional Neural Network involve sliding a filter over an input image and calculating dot product of filter wights and image pixels. To speed up this process, we can distribute computation across multiple units. Tiled shared memory convolution optimizes the transfer of data between shared memory and parallel architecture. It is effective on large datasets and complex models to effectively boost the performance by reducing frequent communication between units. It is widely used in video analytics, robotics and autonomous vehicles which makes it a renowned technique in the field of parallel programming.

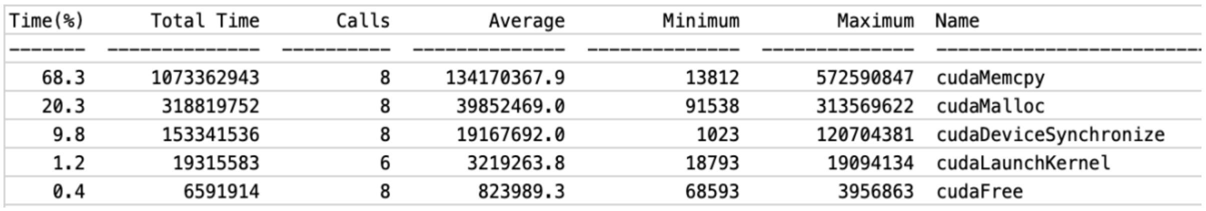
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

The Tiled shared memory convolution is a method of optimizing the forward convolution operation in CNNs. This technique involves dividing the input data into smaller tiles that can fit into the shared memory of processing units. Each processing unit then performs the convolution operation on its assigned tile of the input data using its local weights. The results from all processing units are combined to obtain the final output. This approach reduces the amount of data that needs to be transferred between processing units, leading to a more efficient convolution operation. This technique has been implemented independently and it is a stand-alone procedure when compared to any previous 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).

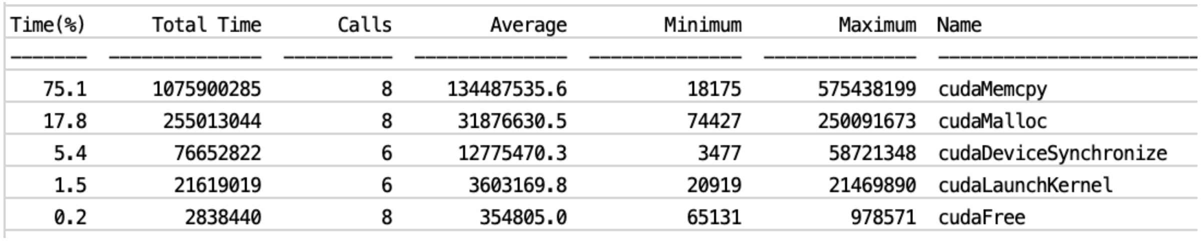
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.340666 ms*	*1.308 ms*	*0m1.265s*	*0.86*
1000	*3.25756 ms*	*12.8584 ms*	*0m9.754s*	*0.886*
10000	*32.0585 ms*	*127.336 ms*	*1m36.487s*	*0.8714*

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



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   68.3      1073362943           8     134170367.9           13812       572590847  cudaMemcpy
   20.3       318819752           8      39852469.0           91538       313569622  cudaMalloc
    9.8       153341536           8      19167692.0            1023       120704381  cudaDeviceSynchronize
    1.2        19315583           6       3219263.8           18793        19094134  cudaLaunchKernel
    0.4         6591914           8        823989.3           68593         3956863  cudaFree
```

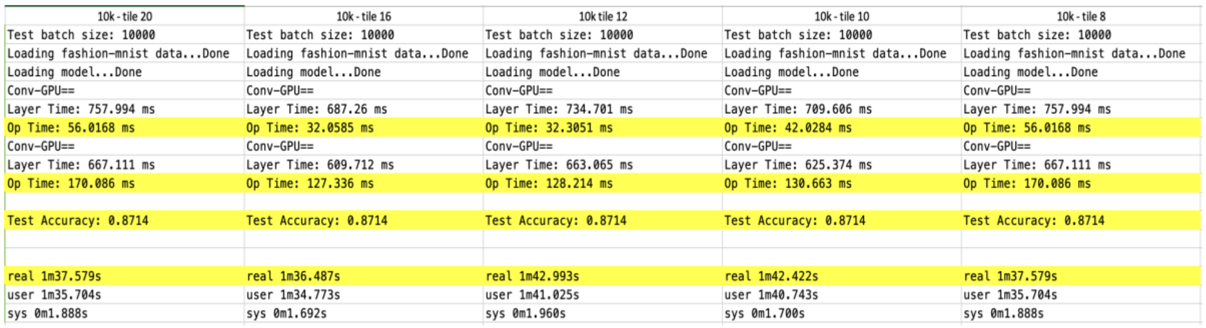
Fig 1a. CUDA API Statistics (nanoseconds) for Tiled shared memory convolution for batch size 10k



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   75.1      1075900285           8     134487535.6           18175       575438199  cudaMemcpy
   17.8       255013044           8      31876630.5           74427       250091673  cudaMalloc
    5.4        76652822           6      12775470.3            3477        58721348  cudaDeviceSynchronize
    1.5        21619019           6       3603169.8           20919        21469890  cudaLaunchKernel
    0.2         2838440           8        354805.0           65131          978571  cudaFree
```

Fig 1b. CUDA API Statistics (nanoseconds) for Baseline for batch size 10k

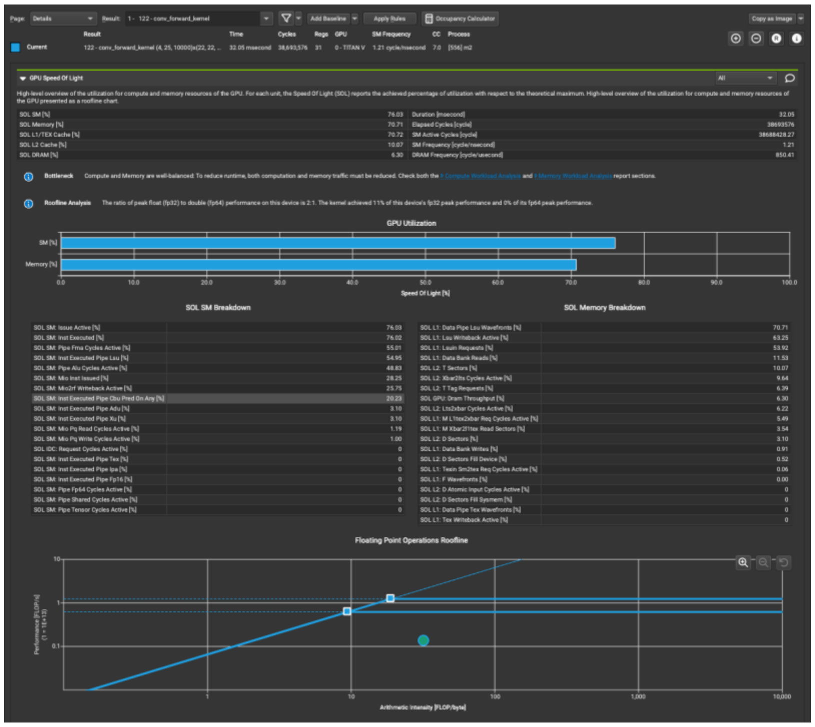
Tiled shared memory convolution is able to reduce the total time of execution of the code by almost 3 seconds on 10K batch size as compared to baseline, even when the batch size is 1K, the optimized technique seems to perform better than the baseline in terms of the total execution time. Furthermore, in the baseline code, 75.1% of time went into copying data from memory using CudaMemcpy which was significantly reduced to 68.3% in tiled shared memory convolution strategy as shown in figure 1a and 1b. On the contrary, the OP times for the tiled shared memory convolution technique have increased to double the value of baseline. This might be because of the tile size, a smaller tile size leads to overhead in transferring the data to and from shared memory whereas a larger tile size might not fit in the memory causing slower performance. Input size and size of shared memory are some other factors that determine the performance and efficiency of the technique.



10k - tile 20	10k - tile 16	10k tile 12	10k - tile 10	10k - tile 8
Test batch size: 10000	Test batch size: 10000	Test batch size: 10000	Test batch size: 10000	Test batch size: 10000
Loading fashion-mnist data...Done	Loading fashion-mnist data...Done	Loading fashion-mnist data...Done	Loading fashion-mnist data...Done	Loading fashion-mnist data...Done
Loading model...Done	Loading model...Done	Loading model...Done	Loading model...Done	Loading model...Done
Conv-GPU==	Conv-GPU==	Conv-GPU==	Conv-GPU==	Conv-GPU==
Layer Time: 757.994 ms	Layer Time: 687.26 ms	Layer Time: 734.701 ms	Layer Time: 709.606 ms	Layer Time: 757.994 ms
Op Time: 56.0168 ms	Op Time: 32.0585 ms	Op Time: 32.3051 ms	Op Time: 42.0284 ms	Op Time: 56.0168 ms
Conv-GPU==	Conv-GPU==	Conv-GPU==	Conv-GPU==	Conv-GPU==
Layer Time: 667.111 ms	Layer Time: 609.712 ms	Layer Time: 663.065 ms	Layer Time: 625.374 ms	Layer Time: 667.111 ms
Op Time: 170.086 ms	Op Time: 127.336 ms	Op Time: 128.214 ms	Op Time: 130.663 ms	Op Time: 170.086 ms
Test Accuracy: 0.8714	Test Accuracy: 0.8714	Test Accuracy: 0.8714	Test Accuracy: 0.8714	Test Accuracy: 0.8714
real 1m37.579s	real 1m36.487s	real 1m42.993s	real 1m42.422s	real 1m37.579s
user 1m35.704s	user 1m34.773s	user 1m41.025s	user 1m40.743s	user 1m35.704s
sys 0m1.888s	sys 0m1.692s	sys 0m1.960s	sys 0m1.700s	sys 0m1.888s

Fig1c. Comparison of run time for batch size 10k and varied tile sizes using tiled shared memory convolution

It is observed from fig 1c that larger tile size and smaller tile size both have negative impact on the performance as the OP time increase as well as the total execution time.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 25, 10000)x(22, 22, ...	32.05 msecond	38,693,576	31	0 - TITAN V	1.21 cycle/nsecond	7.0	[556] m2

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 [%]	76.03	Duration [msecond]	32.05
SOL Memory [%]	70.71	Elapsed Cycles [cycle]	38693576
SOL L1/TEX Cache [%]	70.72	SM Active Cycles [cycle]	38688428.27
SOL L2 Cache [%]	10.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	6.30	DRAM Frequency [cycle/usecond]	850.41

Bottleneck Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis report sections.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

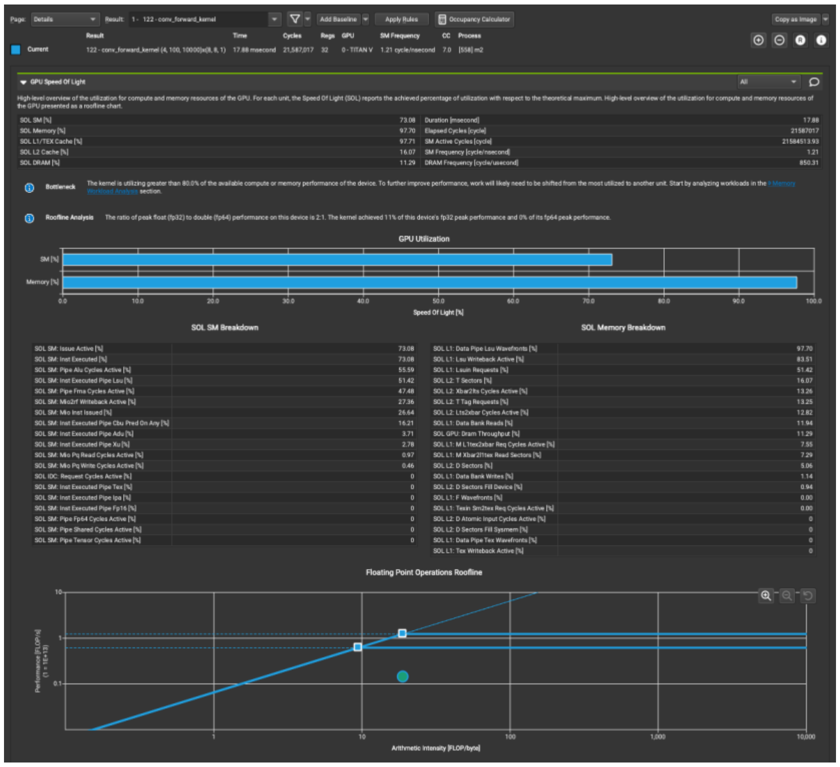
SOL SM Breakdown

SOL SM: Issue Active [%]	76.03
SOL SM: Inst Executed [%]	76.02
SOL SM: Pipe Fma Cycles Active [%]	55.01
SOL SM: Inst Executed Pipe Lsu [%]	54.95
SOL SM: Pipe Alu Cycles Active [%]	48.83
SOL SM: Mio Inst Issued [%]	28.25
SOL SM: Mio2rf Writeback Active [%]	25.75
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	20.23
SOL SM: Inst Executed Pipe Adu [%]	3.10
SOL SM: Inst Executed Pipe Xu [%]	3.10
SOL SM: Mio Pq Read Cycles Active [%]	1.19
SOL SM: Mio Pq Write Cycles Active [%]	1.00
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	70.71
SOL L1: Lsu Writeback Active [%]	63.25
SOL L1: Lsuin Requests [%]	53.92
SOL L1: Data Bank Reads [%]	11.53
SOL L2: T Sectors [%]	10.07
SOL L2: Xbar2lts Cycles Active [%]	9.64
SOL L2: T Tag Requests [%]	6.39
SOL GPU: Dram Throughput [%]	6.30
SOL L2: Lts2xbar Cycles Active [%]	6.22
SOL L1: M L1tex2xbar Req Cycles Active [%]	5.49
SOL L1: M Xbar2l1tex Read Sectors [%]	3.54
SOL L2: D Sectors [%]	3.10
SOL L1: Data Bank Writes [%]	0.91
SOL L2: D Sectors Fill Device [%]	0.52
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.06
SOL L1: F Wavefronts [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 1d. Analysis of Tiled shared memory convolution technique using Nsight-Compute for batch size 10k



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	17.88 msecond	21,587,017	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[558] m2

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 [%]	73.08	Duration [msecond]	17.88
SOL Memory [%]	97.70	Elapsed Cycles [cycle]	21587017
SOL L1/TEX Cache [%]	97.71	SM Active Cycles [cycle]	21584513.93
SOL L2 Cache [%]	16.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.29	DRAM Frequency [cycle/usecond]	850.31

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	73.08
SOL SM: Inst Executed [%]	73.08
SOL SM: Pipe Alu Cycles Active [%]	55.59
SOL SM: Inst Executed Pipe Lsu [%]	51.42
SOL SM: Pipe Fma Cycles Active [%]	47.48
SOL SM: Mio2rf Writeback Active [%]	27.36
SOL SM: Mio Inst Issued [%]	26.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.21
SOL SM: Inst Executed Pipe Adu [%]	3.71
SOL SM: Inst Executed Pipe Xu [%]	2.78
SOL SM: Mio Pq Read Cycles Active [%]	0.97
SOL SM: Mio Pq Write Cycles Active [%]	0.46
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	97.70
SOL L1: Lsu Writeback Active [%]	83.51
SOL L1: Lsuin Requests [%]	51.42
SOL L2: T Sectors [%]	16.07
SOL L2: Xbar2lts Cycles Active [%]	13.26
SOL L2: T Tag Requests [%]	13.25
SOL L2: Lts2xbar Cycles Active [%]	12.82
SOL L1: Data Bank Reads [%]	11.94
SOL GPU: Dram Throughput [%]	11.29
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.55
SOL L1: M Xbar2l1tex Read Sectors [%]	7.29
SOL L2: D Sectors [%]	5.06
SOL L1: Data Bank Writes [%]	1.14
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 1e. Analysis of baseline technique using Nsight-Compute for batch size 10k

While comparing the Nsight-Compute results, it is observed that the baseline code utilized 97% of the compute or memory performance of the device which is well balanced at around 70% for the newly implemented technique of Tiled *shared memory convolution as shown in figure 1d and 1e. Additionally, the arithmetic intensity (FLOP/byte) has increased in tiled shared memory convolution technique as seen in the floating-point operations roofline graph. Overall, the tiled shared memory convolution technique performs better than the baseline in several aspects such as memory utilization, arithmetic intensity etc.*

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

1. https://forums.developer.nvidia.com/t/tiled-2d-convolution-algorithm-as-slow-as-untiled-2d-convolution-algorithm/164862/3

2. https://lumetta.web.engr.illinois.edu/408-S19/slide-copies/ece408-lecture8-S19-ZJUI.pdf

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"

#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)
{
    /*
    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; // Compute the output height and width dimensions
  const int Width_out = Width - K + 1;
  const int Width_grid = (Width_out + Tile_Width - 1) / Tile_Width; // Compute the number of blocks required for the output width
dimension
  int Blk_width = Tile_Width + K - 1; // Compute the block width to accommodate the kernel

  extern __shared__ float Shared_Mem[]; // Allocate shared memory
  float* Shared_Mem_obj = &Shared_Mem[0]; // Create a pointer to the start of the shared memory block

#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 bidz = blockIdx.z;
  int bidx = blockIdx.x;
  int bidy = blockIdx.y;
  int t = threadIdx.x + threadIdx.y * Blk_width;
  int height = (bidy / Width_grid) * Tile_Width + threadIdx.y; // Compute the input height and width indices
  int width = (bidy % Width_grid) * Tile_Width + threadIdx.x;
  float inter = 0.0f;

  for(int c = 0; c < Channel; c++){
    // copy data from global memory to shared memory
    if((height < Height) && (width < Width))
      Shared_Mem_obj[t] = in_4d(bidz, c, height, width); // Copy data from global memory to shared memory
    else
      Shared_Mem_obj[t] = 0.0f;
    __syncthreads();

    // convolution
    if((height < Height_out) && (width < Width_out)){
      for(int p = 0; p < K; p++)
        for(int q = 0; q < K; q++)
          inter += in_4d(bidz, c, height + p, width + q) * mask_4d(bidx, c, p, q);
    }
    __syncthreads();
  }
  if((height < Height_out) && (width < Width_out))
```

```
        out_4d(bidz, bidx, height, width) = inter;

#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 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
    const int Height_out = Height - K + 1; // Calculate the output dimensions
    const int Width_out = Width - K + 1;
    int value =  Map_out * Channel * K * K * sizeof(float); // Calculate the sizes of the relevant data structures
    int value_2 = Batch * Channel * Height * Width * sizeof(float);
    int value_3 = Batch * Map_out * Height_out * Width_out * sizeof(float);

    cudaMalloc((void**)device_output_ptr, value_3); // Allocate memory on the GPU for the output, input, and mask
    cudaMalloc((void**)device_input_ptr, value_2);
    cudaMalloc((void**)device_mask_ptr,value);

    cudaMemcpy(*device_input_ptr, host_input, value_2, cudaMemcpyHostToDevice); // Copy the input and mask data from the host to the
device
    cudaMemcpy(*device_mask_ptr, host_mask, value, cudaMemcpyHostToDevice);

}


__host__ void GPUInterface::conv_forward_gpu(float *device_y, const float *device_x, const float *device_k, const int Batch, const int
Map_out, const int Channel, const int Height, const int Width, const int K)
{
    // Set the kernel dimensions and call the kernel
    const int Height_ = ((Height - K + 1) + Tile_Width - 1) / Tile_Width; // Calculate the dimensions of the kernel block and grid
    const int Width_ = ((Width - K + 1) + Tile_Width - 1) / Tile_Width;
    int Blk_width = Tile_Width + K - 1;

    dim3 blockDim(Blk_width, Blk_width, 1); // Set the dimensions of the kernel block and grid
    dim3 gridDim(Map_out, Width_ * Height_ , Batch);

    size_t Shared_Mem_obj1 = (Blk_width) * (Blk_width) * sizeof(float); // Allocate shared memory for the kernel
    conv_forward_kernel<<<gridDim, blockDim, Shared_Mem_obj1>>>(device_y, device_x, device_k, Batch, Map_out, Channel, Height, 
Width, K);
    cudaDeviceSynchronize();
}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_y, float *device_x, float *device_k, const int Batch, 
const int Map_out, const int Channel, const int Height, const int Width, const int K)
{
    // Copy the output back to host
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    cudaMemcpy(host_output, device_y, Batch * Map_out * Height_out * Width_out * sizeof(float), cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(device_y);
    cudaFree(device_x);
    cudaFree(device_k);
}


__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;
    }
}
```

2. **Optimization 2: Weight matrix in constant memory**

   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

   The weight matrix in constant memory optimization technique improves the efficiency of parallel programming by storing the weight matrix in constant memory. This enables the processing units to perform multiple convolution operations without the need to reload the weight matrix each time, resulting in faster computation times and better overall performance.

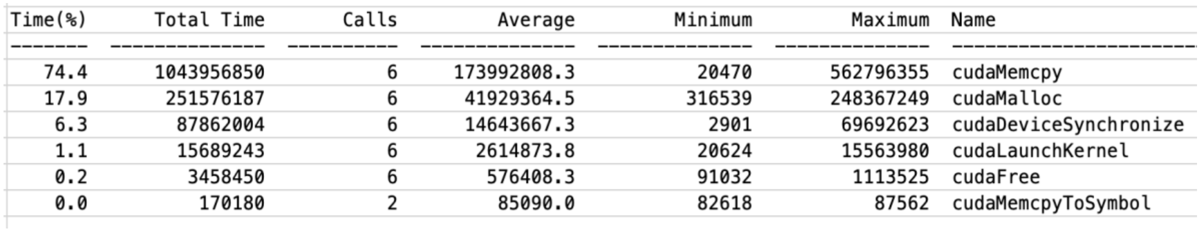
   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

   To enhance the efficiency of the convolution operation in parallel programming, the weight matrix in constant memory optimization technique involves storing the weight matrix in a type of memory called constant memory. Constant memory can be swiftly accessed by all processing units in a parallel system, thus minimizing communication between them and improving performance. By storing the weight matrix in constant memory, processing units can be utilized better, allowing them to carry out multiple convolution operations without needing to reload the weight matrix every time. Thus, improving the performance of the forward convolution. This technique has been implemented independently and it is a stand-alone procedure when compared to any previous 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).

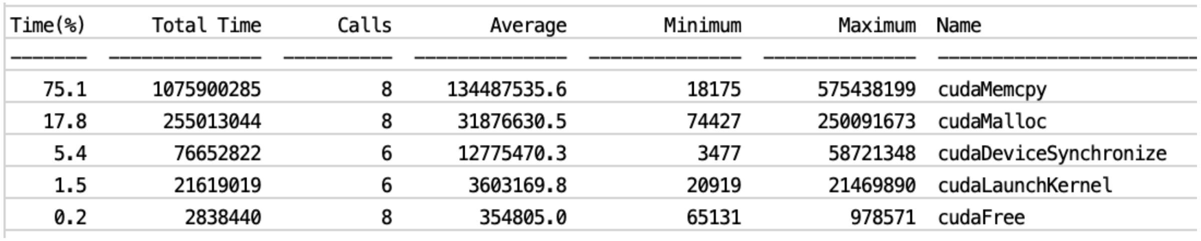
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.197887 ms*	*0.735227 ms*	*0m1.248s*	*0.86*
1000	*1.82382 ms*	*7.09912 ms*	*0m9.713s*	*0.886*
10000	*18.6088 ms*	*70.7365 ms*	*1m44.441s*	*0.8714*

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



```
Time(%)    Total Time      Calls       Average        Minimum        Maximum   Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------
   74.4      1043956850           6     173992808.3           20470       562796355  cudaMemcpy
   17.9       251576187           6      41929364.5          316539       248367249  cudaMalloc
    6.3        87862004           6      14643667.3            2901        69692623  cudaDeviceSynchronize
    1.1        15689243           6       2614873.8           20624        15563980  cudaLaunchKernel
    0.2         3458450           6        576408.3           91032         1113525  cudaFree
    0.0          170180           2         85090.0           82618           87562  cudaMemcpyToSymbol
```

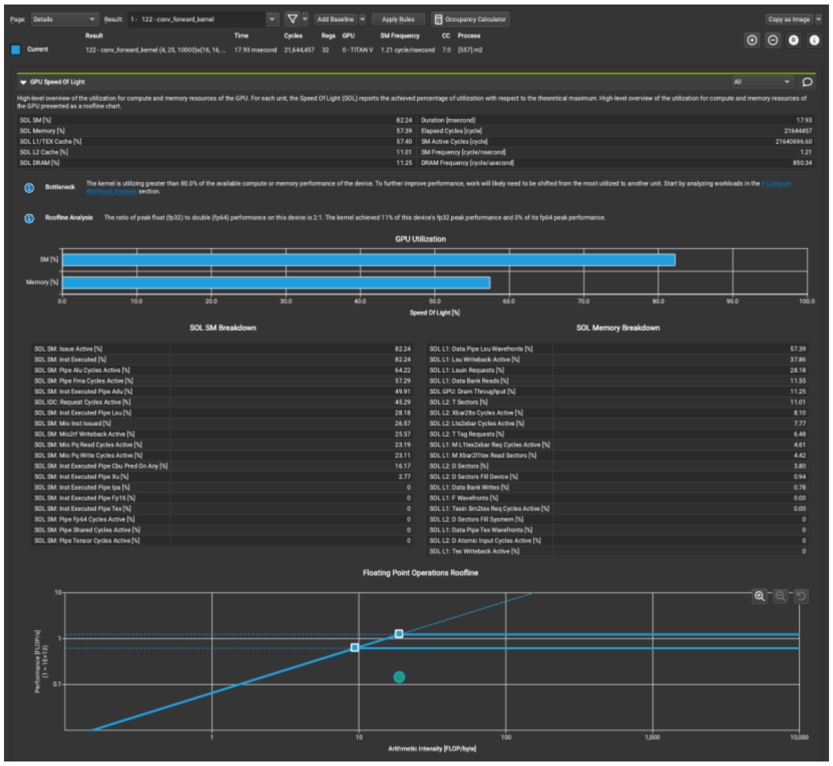
Fig 2a. CUDA API Statistics (nanoseconds) for weight matrix in constant memory optimization technique for batch size 10k



```
Time(%)    Total Time      Calls       Average        Minimum        Maximum   Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------
   75.1      1075900285           8     134487535.6           18175       575438199  cudaMemcpy
   17.8       255013044           8      31876630.5           74427       250091673  cudaMalloc
    5.4        76652822           6      12775470.3            3477        58721348  cudaDeviceSynchronize
    1.5        21619019           6       3603169.8           20919        21469890  cudaLaunchKernel
    0.2         2838440           8        354805.0           65131          978571  cudaFree
```

Fig 2b. CUDA API Statistics (nanoseconds) for Baseline for batch size 10k

Weight matrix in constant memory optimization technique has performed equally well as the baseline technique. There is a similar CUDA API usage and CUDA Kernel statistics, however, on a batch size of 10K, the optimized technique tends to be 5 seconds slower as shown in fig 2a and fig 2b. Furthermore, the OP time of layer 2 has significantly increased by 12 msec when compared to the baseline technique. However, there are multiple reasons why this technique could not improve the performance, it might be because of limited constant memory.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 25, 10000)x(16, 16, ...	17.93 msecond	21,644,457	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[557] m2

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 [%]	82.24	Duration [msecond]	17.93
SOL Memory [%]	57.39	Elapsed Cycles [cycle]	21644457
SOL L1/TEX Cache [%]	57.40	SM Active Cycles [cycle]	21640696.60
SOL L2 Cache [%]	11.01	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.25	DRAM Frequency [cycle/usecond]	850.34

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

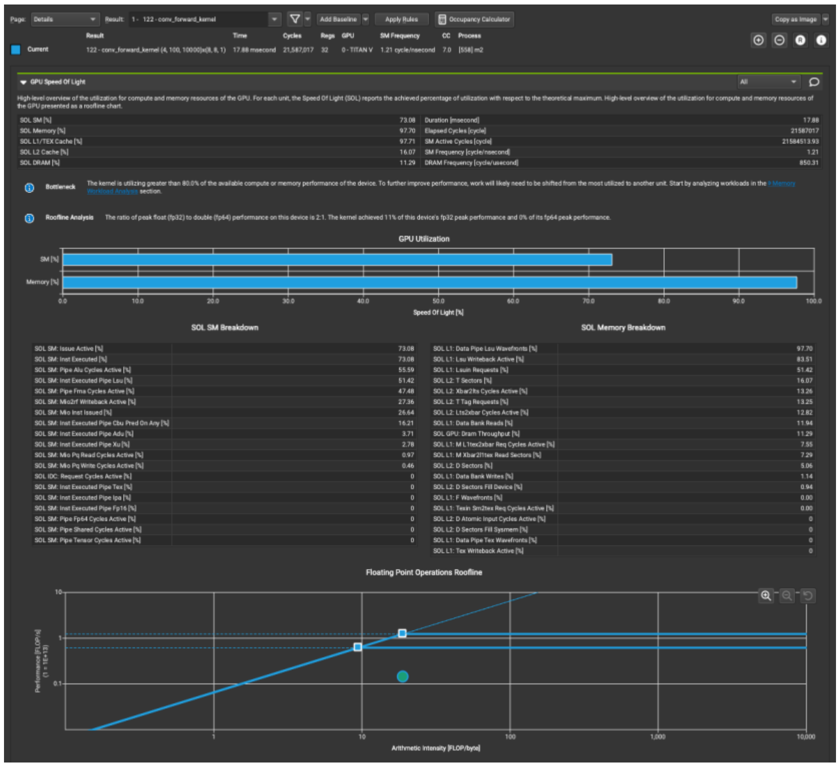
SOL SM Breakdown

SOL SM: Issue Active [%]	82.24
SOL SM: Inst Executed [%]	82.24
SOL SM: Pipe Alu Cycles Active [%]	64.22
SOL SM: Pipe Fma Cycles Active [%]	57.29
SOL SM: Inst Executed Pipe Adu [%]	49.91
SOL IDC: Request Cycles Active [%]	45.29
SOL SM: Inst Executed Pipe Lsu [%]	28.18
SOL SM: Mio Inst Issued [%]	26.57
SOL SM: Mio2rf Writeback Active [%]	25.57
SOL SM: Mio Pq Read Cycles Active [%]	23.19
SOL SM: Mio Pq Write Cycles Active [%]	23.11
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.17
SOL SM: Inst Executed Pipe Xu [%]	2.77
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	57.39
SOL L1: Lsu Writeback Active [%]	37.86
SOL L1: Lsuin Requests [%]	28.18
SOL L1: Data Bank Reads [%]	11.55
SOL GPU: Dram Throughput [%]	11.25
SOL L2: T Sectors [%]	11.01
SOL L2: Xbar2lts Cycles Active [%]	8.10
SOL L2: Lts2xbar Cycles Active [%]	7.77
SOL L2: T Tag Requests [%]	6.48
SOL L1: M L1tex2xbar Req Cycles Active [%]	4.61
SOL L1: M Xbar2l1tex Read Sectors [%]	4.42
SOL L2: D Sectors [%]	3.80
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: Data Bank Writes [%]	0.78
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 2c. Analysis of weight matrix in constant memory optimization technique using Nsight-Compute for batch size 10k



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	17.88 msecond	21,587,017	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[558] m2

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 [%]	73.08	Duration [msecond]	17.88
SOL Memory [%]	97.70	Elapsed Cycles [cycle]	21587017
SOL L1/TEX Cache [%]	97.71	SM Active Cycles [cycle]	21584513.93
SOL L2 Cache [%]	16.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.29	DRAM Frequency [cycle/usecond]	850.31

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	73.08
SOL SM: Inst Executed [%]	73.08
SOL SM: Pipe Alu Cycles Active [%]	55.59
SOL SM: Inst Executed Pipe Lsu [%]	51.42
SOL SM: Pipe Fma Cycles Active [%]	47.48
SOL SM: Mio2rf Writeback Active [%]	27.36
SOL SM: Mio Inst Issued [%]	26.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.21
SOL SM: Inst Executed Pipe Adu [%]	3.71
SOL SM: Inst Executed Pipe Xu [%]	2.78
SOL SM: Mio Pq Read Cycles Active [%]	0.97
SOL SM: Mio Pq Write Cycles Active [%]	0.46
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	97.70
SOL L1: Lsu Writeback Active [%]	83.51
SOL L1: Lsuin Requests [%]	51.42
SOL L2: T Sectors [%]	16.07
SOL L2: Xbar2lts Cycles Active [%]	13.26
SOL L2: T Tag Requests [%]	13.25
SOL L2: Lts2xbar Cycles Active [%]	12.82
SOL L1: Data Bank Reads [%]	11.94
SOL GPU: Dram Throughput [%]	11.29
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.55
SOL L1: M Xbar2l1tex Read Sectors [%]	7.29
SOL L2: D Sectors [%]	5.06
SOL L1: Data Bank Writes [%]	1.14
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 2d. Analysis of baseline technique using Nsight-Compute for batch size 10k

While analyzing and comparing the Nsight-Compute of weight matrix in constant memory optimization and baseline technique, it is observed that the GPU utilization of Streaming Multiprocessor (SM) increased from 73% (baseline) to 82% (weight matrix in constant memory). Furthermore, there is a significant 40% decline in the memory performance, dropping from 97% (baseline) to 57% (weight matrix in constant memory) as shown in fig 2c and fig 2d. The arithmetic intensity in both the cases remains almost the same. Although the OP times, accuracy, and total time of execution did not improve much, the GPU and memory performance has enhanced.

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

1. *https://www.sciencedirect.com/science/article/pii/S1319157820304845*
2. *https://quasar.ugent.be/files/doc/CUDA-Constant-Memory.html*
3. *https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3262956/*
4. *https://developer.download.nvidia.com/GTC/PDF/1083_Wang.pdf*
5. *https://passlab.github.io/CSCE790/notes/lecture12_GPUArchCUDA02-CUDAMem.pptx*

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"

#define TILE_WIDTH 16
__constant__ float const_mem_matrix[15000];

__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;// Define output height and width
    const int Width_out = Width - K + 1;

    const int Width_grid = (Width_out + TILE_WIDTH - 1) / TILE_WIDTH; // Compute number of thread blocks needed to cover the output tensor

    // (void)Height_out; // silence declared but never referenced warning. remove this line when you start working
    // (void)Width_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) * (Map_out * Height_out * Width_out) + (i2) * (Height_out * Width_out) + (i1) * (Width_out) + i0] // Define macros to simplify indexing of tensors
#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) const_mem_matrix[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here

    int H = (blockIdx.y / Width_grid) * TILE_WIDTH + threadIdx.y; // Compute the index of the pixel to be computed by the current thread
    int W = (blockIdx.y % Width_grid) * TILE_WIDTH + threadIdx.x;

    // Check if the thread is within the bounds of the output tensor
    if (H < Height_out && W < Width_out){ // for all height and width pixel values
        float inter = 0.0f; // declaring a temp variable
        for (int C = 0; C < Channel; C++) { // sum over all channels
            for (int k = 0; k < K; k++){ // loop over KxK filter
                for (int i = 0; i < K; i++){
                    // Compute the dot product of the input tensor and filter kernel
                    inter += in_4d(blockIdx.z, C, H + k, W + i) * mask_4d(blockIdx.x, C, k, i); // calculating convolution and adding the intermediate results to inter variable
                }
            }
        }
        out_4d(blockIdx.z, blockIdx.x, H, W) = inter; // storing the final results in out_4d
    }

    #undef out_4d // Undefine macros
    #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 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

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

    int out_size = ((Height_out*Width_out) * Map_out * Batch) * sizeof(float); // output size is batchsize * output channels * size of each output image
    int in_size = (Height*Width) * Channel * Batch * sizeof(float); // input size is input image dimensions * channels * batchsize
    int k_size = (K*K) * Map_out * Channel * sizeof(float); //each filter times input channels and output feature maps

    cudaMalloc((void**)device_input_ptr, in_size);
    // cudaMalloc((void**)device_mask_ptr, k_size);
    cudaMalloc((void**)device_output_ptr, out_size);


    cudaMemcpy(*device_input_ptr, host_input, in_size, cudaMemcpyHostToDevice);
    // cudaMemcpy(*device_mask_ptr, host_mask, k_size, cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(const_mem_matrix, host_mask, k_size);

}


__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 Channel, const int Height, const int Width, const int K)
{
    // Set the kernel dimensions and call the kernel
    const int Height_ = ((Height - K + 1) + TILE_WIDTH - 1) / TILE_WIDTH; // Calculate the dimensions of the kernel block and grid
    const int Width_ = ((Width - K + 1) + TILE_WIDTH - 1) / TILE_WIDTH;

    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1); // Set the dimensions of the kernel block and grid
    dim3 gridDim(Map_out, Width_ * Height_ , Batch);

    conv_forward_kernel<<<gridDim, blockDim>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);

}
__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_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 out_size = (Height_out*Width_out) * Map_out * Batch * sizeof(float);

    // Copy the output back to host

    cudaMemcpy(host_output, device_output, out_size, cudaMemcpyDeviceToHost);

    // Free device memory

    cudaFree(device_output);
    cudaFree(device_input);
    // cudaFree(device_mask);
}
__host__ void GPUInterface::get_device_properties()
{
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

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

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

3. **Optimization 3: Tuning with restrict and loop unrolling**

   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

Tuning with restrict and loop unrolling is used in parallel programming to optimize the performance of code that includes looping and utilizes shared memory. The restrict keyword informs the compiler that the memory accessed by a pointer is unique and not shared with any other pointer, allowing the compiler to generate more efficient code. Loop unrolling reduces the overhead of loop control structures, enabling more efficient memory access patterns and reducing the number of instructions executed by the program. Together, these techniques can result in significant performance improvements for parallel code.

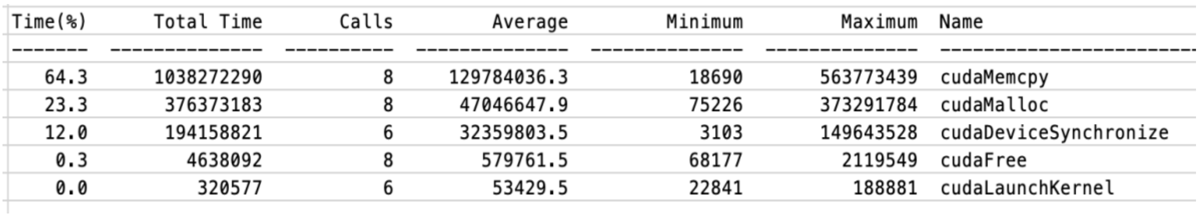
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

The Tuning with restrict and loop unrolling technique leverages a keyword called "restrict" which denotes that a pointer is not aliased. When a pointer is not aliased, it does not point to the same memory location. This benefits the compiler in generating efficient code by making assumption on memory access patterns. It only requires the keyword "restrict" to be added before the pointer declaration. Loops are a portion of the code which are highly benefited by this technique. Looping unrolling involves replicating the body of the loop multiple times such that in each occurrence it works on a different subset of data. This reduces overhead with loop control and make efficient use of vector instructions. Overall, this technique increases the performance by efficiently performing memory access and parallelly working on different subset of the large dataset in every single run of the loop. This technique has been implemented independently and it is a stand-alone procedure when compared to any previous 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).

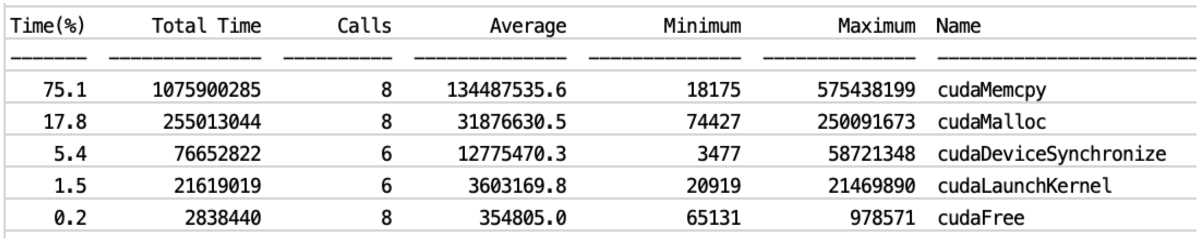
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.21463 ms*	*0.657543 ms*	*0m1.175s*	*0.86*
1000	*1.98175 ms*	*6.38672 ms*	*0m10.151s*	*0.876*
10000	*19.5965 ms*	*63.672 ms*	*1m39.450s*	*0.8747*

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



```
Time(%)    Total Time     Calls       Average        Minimum        Maximum   Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   64.3      1038272290           8     129784036.3           18690       563773439  cudaMemcpy
   23.3       376373183           8      47046647.9           75226       373291784  cudaMalloc
   12.0       194158821           6      32359803.5            3103       149643528  cudaDeviceSynchronize
    0.3         4638092           8        579761.5           68177         2119549  cudaFree
    0.0          320577           6         53429.5           22841          188881  cudaLaunchKernel
```

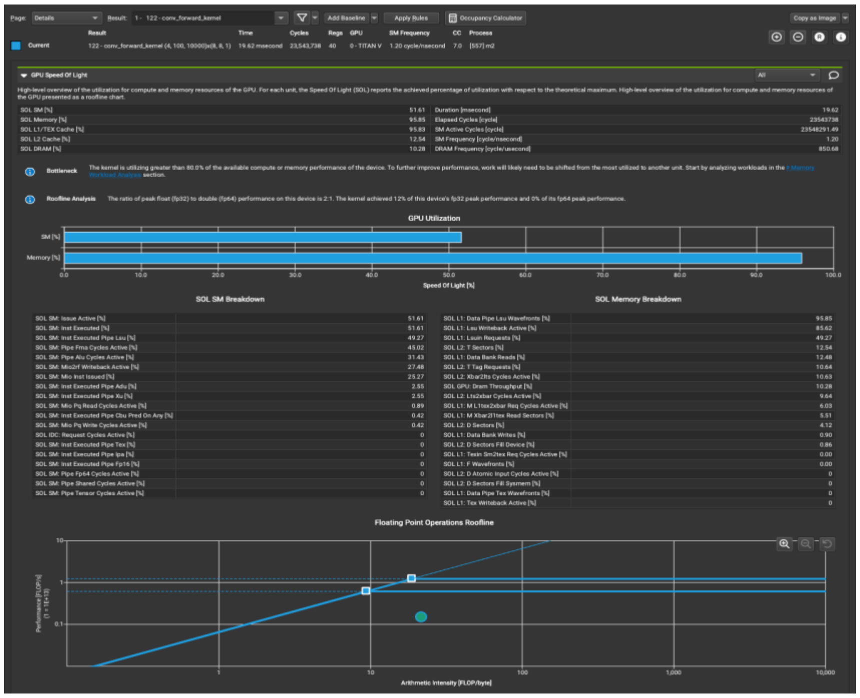
Fig 3a. CUDA API Statistics (nanoseconds) for Tuning with restrict and loop unrolling technique for batch size 10k



```
Time(%)    Total Time     Calls       Average        Minimum        Maximum   Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   75.1      1075900285           8     134487535.6           18175       575438199  cudaMemcpy
   17.8       255013044           8      31876630.5           74427       250091673  cudaMalloc
    5.4        76652822           6      12775470.3            3477        58721348  cudaDeviceSynchronize
    1.5        21619019           6       3603169.8           20919        21469890  cudaLaunchKernel
    0.2         2838440           8        354805.0           65131          978571  cudaFree
```

Fig 3b. CUDA API Statistics (nanoseconds) for Baseline for batch size 10k

The implementation of tuning with restrict and loop unrolling technique has performed as good as the baseline technique across all the 3 batch sizes. Additionally, one major upgrade is the accuracy on 10K batch size which increased from 0.8714 to 0.8747. Furthermore, there was a decrease of 10% of time spend in copying data to and from memory. There was an increase in the time consumed in cudaMalloc and cudaDeviceSynchronize API as shown in fig 3a and 3b. Factors that affect the performance are the tile size and number of repetitions in the loop.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	19.62 msecond	23,543,738	40	0 - TITAN V	1.20 cycle/nsecond	7.0	[557] m2

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 [%]	51.61	Duration [msecond]	19.62
SOL Memory [%]	95.85	Elapsed Cycles [cycle]	23543738
SOL L1/TEX Cache [%]	95.83	SM Active Cycles [cycle]	23548291.49
SOL L2 Cache [%]	12.54	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	10.28	DRAM Frequency [cycle/usecond]	850.68

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 12% of this device's fp32 peak performance and 0% of its fp64 peak performance.

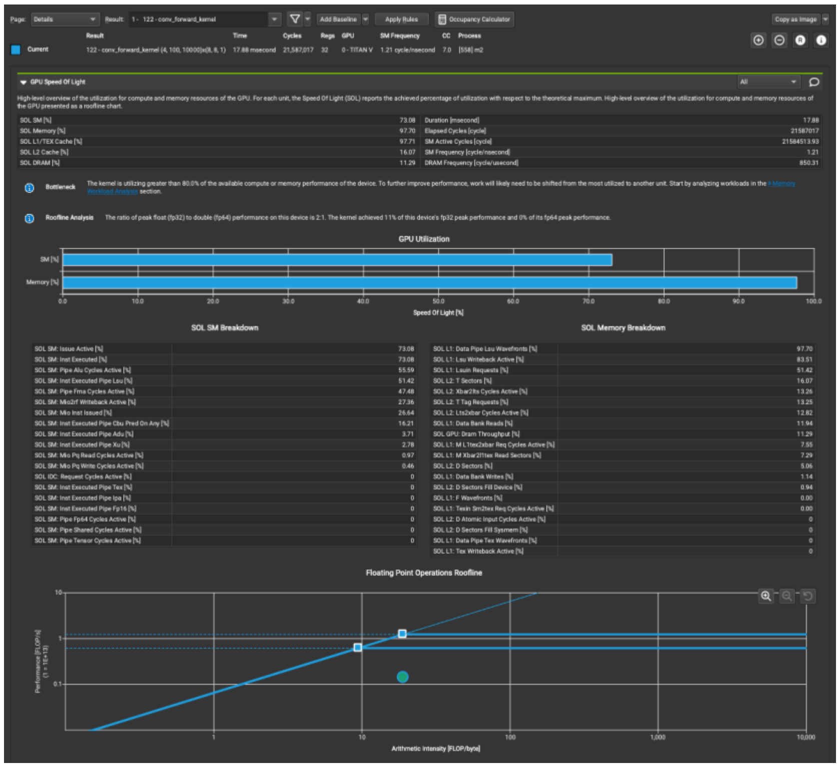
SOL SM Breakdown

SOL SM: Issue Active [%]	51.61
SOL SM: Inst Executed [%]	51.61
SOL SM: Inst Executed Pipe Lsu [%]	49.27
SOL SM: Pipe Fma Cycles Active [%]	45.02
SOL SM: Pipe Alu Cycles Active [%]	31.43
SOL SM: Mio2rf Writeback Active [%]	27.48
SOL SM: Mio Inst Issued [%]	25.27
SOL SM: Inst Executed Pipe Adu [%]	2.55
SOL SM: Inst Executed Pipe Xu [%]	2.55
SOL SM: Mio Pq Read Cycles Active [%]	0.89
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	0.42
SOL SM: Mio Pq Write Cycles Active [%]	0.42
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	95.85
SOL L1: Lsu Writeback Active [%]	85.62
SOL L1: Lsuin Requests [%]	49.27
SOL L2: T Sectors [%]	12.54
SOL L1: Data Bank Reads [%]	12.48
SOL L2: T Tag Requests [%]	10.64
SOL L2: Xbar2lts Cycles Active [%]	10.63
SOL GPU: Dram Throughput [%]	10.28
SOL L2: Lts2xbar Cycles Active [%]	9.64
SOL L1: M L1tex2xbar Req Cycles Active [%]	6.03
SOL L1: M Xbar2l1tex Read Sectors [%]	5.51
SOL L2: D Sectors [%]	4.12
SOL L1: Data Bank Writes [%]	0.90
SOL L2: D Sectors Fill Device [%]	0.86
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L1: F Wavefronts [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 3c. Analysis of tuning with restrict and loop unrolling technique using Nsight-Compute for batch size 10k



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	17.88 msecond	21,587,017	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[558] m2

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 [%]	73.08	Duration [msecond]	17.88
SOL Memory [%]	97.70	Elapsed Cycles [cycle]	21587017
SOL L1/TEX Cache [%]	97.71	SM Active Cycles [cycle]	21584513.93
SOL L2 Cache [%]	16.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.29	DRAM Frequency [cycle/usecond]	850.31

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	73.08
SOL SM: Inst Executed [%]	73.08
SOL SM: Pipe Alu Cycles Active [%]	55.59
SOL SM: Inst Executed Pipe Lsu [%]	51.42
SOL SM: Pipe Fma Cycles Active [%]	47.48
SOL SM: Mio2rf Writeback Active [%]	27.36
SOL SM: Mio Inst Issued [%]	26.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.21
SOL SM: Inst Executed Pipe Adu [%]	3.71
SOL SM: Inst Executed Pipe Xu [%]	2.78
SOL SM: Mio Pq Read Cycles Active [%]	0.97
SOL SM: Mio Pq Write Cycles Active [%]	0.46
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	97.70
SOL L1: Lsu Writeback Active [%]	83.51
SOL L1: Lsuin Requests [%]	51.42
SOL L2: T Sectors [%]	16.07
SOL L2: Xbar2lts Cycles Active [%]	13.26
SOL L2: T Tag Requests [%]	13.25
SOL L2: Lts2xbar Cycles Active [%]	12.82
SOL L1: Data Bank Reads [%]	11.94
SOL GPU: Dram Throughput [%]	11.29
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.55
SOL L1: M Xbar2l1tex Read Sectors [%]	7.29
SOL L2: D Sectors [%]	5.06
SOL L1: Data Bank Writes [%]	1.14
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 3d. Analysis of baseline technique using Nsight-Compute for batch size 10k

As observed using Nsight-Compute, the tuning with restrict and loop unrolling technique has 20% lower SM consumption with a similar percentage of memory *consumption. Additionally, the arithmetic intensity [FLOP/byte] is slightly higher for the optimized technique over baseline as shown in fig 3c and fig 3d.*

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

1. *https://hpc.ac.upc.edu/PDFs/dir00/file003683.pdf*
2. *https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=osu1253131903&disposition=inline*
3. *https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf*

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"

#define TILE_WIDTH 8

__global__ void conv_forward_kernel(float* __restrict__ output, const float* __restrict__ input, const float* __restrict__ 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; // Calculate the width and height of each output
    const int Width_out = Width - K + 1;



    // (void)Height_out; // silence declared but never referenced warning. remove this line when you start working
    // (void)Width_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) * (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]
    float var = 0.0f;

    // Insert your GPU convolution kernel code here
    const int Width_grid = (Width_out + TILE_WIDTH - 1) / TILE_WIDTH;
    int height = (blockIdx.y / Width_grid) * TILE_WIDTH + threadIdx.y; // Calculate the height and width indices for the current thread
    int width = (blockIdx.y % Width_grid) * TILE_WIDTH + threadIdx.x;

    if((height < Height_out) && (width < Width_out)){
        for(int c = 0; c < Channel; c++){
            var = var + in_4d(blockIdx.z,c,height+0,width+0) * mask_4d(blockIdx.x,c,0,0) + in_4d(blockIdx.z,c,height+0,width+1) * mask_4d(blockIdx.x,c,0,1) +
in_4d(blockIdx.z,c,height+0,width+2) * mask_4d(blockIdx.x,c,0,2) + in_4d(blockIdx.z,c,height+0,width+3) * mask_4d(blockIdx.x,c,0,3)+
in_4d(blockIdx.z,c,height+0,width+4) * mask_4d(blockIdx.x,c,0,4)+ in_4d(blockIdx.z,c,height+0,width+5) * mask_4d(blockIdx.x,c,0,5)+
in_4d(blockIdx.z,c,height+0,width+6) * mask_4d(blockIdx.x,c,0,6) ;

            var = var + in_4d(blockIdx.z,c,height+1,width+0) * mask_4d(blockIdx.x,c,1,0) + in_4d(blockIdx.z,c,height+1,width+1) * mask_4d(blockIdx.x,c,1,1) +
in_4d(blockIdx.z,c,height+1,width+2) * mask_4d(blockIdx.x,c,1,2) + in_4d(blockIdx.z,c,height+1,width+3) * mask_4d(blockIdx.x,c,1,3)+
in_4d(blockIdx.z,c,height+1,width+4) * mask_4d(blockIdx.x,c,1,4)+ in_4d(blockIdx.z,c,height+1,width+5) * mask_4d(blockIdx.x,c,1,5)+
in_4d(blockIdx.z,c,height+1,width+6) * mask_4d(blockIdx.x,c,1,6) ;

            var = var + in_4d(blockIdx.z,c,height+2,width+0) * mask_4d(blockIdx.x,c,2,0) + in_4d(blockIdx.z,c,height+2,width+1) * mask_4d(blockIdx.x,c,2,1) +
in_4d(blockIdx.z,c,height+2,width+2) * mask_4d(blockIdx.x,c,2,2) + in_4d(blockIdx.z,c,height+2,width+3) * mask_4d(blockIdx.x,c,2,3)+
in_4d(blockIdx.z,c,height+2,width+4) * mask_4d(blockIdx.x,c,2,4)+ in_4d(blockIdx.z,c,height+2,width+5) * mask_4d(blockIdx.x,c,2,5)+
in_4d(blockIdx.z,c,height+2,width+6) * mask_4d(blockIdx.x,c,2,6) ;

            var = var + in_4d(blockIdx.z,c,height+3,width+0) * mask_4d(blockIdx.x,c,3,0) + in_4d(blockIdx.z,c,height+3,width+1) * mask_4d(blockIdx.x,c,3,1) +
in_4d(blockIdx.z,c,height+3,width+2) * mask_4d(blockIdx.x,c,3,2) + in_4d(blockIdx.z,c,height+3,width+3) * mask_4d(blockIdx.x,c,3,3)+
in_4d(blockIdx.z,c,height+3,width+4) * mask_4d(blockIdx.x,c,3,4)+ in_4d(blockIdx.z,c,height+3,width+5) * mask_4d(blockIdx.x,c,3,5)+
in_4d(blockIdx.z,c,height+3,width+6) * mask_4d(blockIdx.x,c,3,6) ;

            var = var + in_4d(blockIdx.z,c,height+4,width+0) * mask_4d(blockIdx.x,c,4,0) + in_4d(blockIdx.z,c,height+4,width+1) * mask_4d(blockIdx.x,c,4,1) +
in_4d(blockIdx.z,c,height+4,width+2) * mask_4d(blockIdx.x,c,4,2) + in_4d(blockIdx.z,c,height+4,width+3) * mask_4d(blockIdx.x,c,4,3)+
in_4d(blockIdx.z,c,height+4,width+4) * mask_4d(blockIdx.x,c,4,4)+ in_4d(blockIdx.z,c,height+4,width+5) * mask_4d(blockIdx.x,c,4,5)+
in_4d(blockIdx.z,c,height+4,width+6) * mask_4d(blockIdx.x,c,4,6);
```

```
            var = var + in_4d(blockIdx.z,c,height+5,width+0) * mask_4d(blockIdx.x,c,5,0) + in_4d(blockIdx.z,c,height+5,width+1) * mask_4d(blockIdx.x,c,5,1) +
in_4d(blockIdx.z,c,height+5,width+2) * mask_4d(blockIdx.x,c,5,2) + in_4d(blockIdx.z,c,height+5,width+3) * mask_4d(blockIdx.x,c,5,3)+
in_4d(blockIdx.z,c,height+5,width+4) * mask_4d(blockIdx.x,c,5,4)+ in_4d(blockIdx.z,c,height+5,width+5) * mask_4d(blockIdx.x,c,5,5)+
in_4d(blockIdx.z,c,height+5,width+6) * mask_4d(blockIdx.x,c,5,6) ;

            var = var + in_4d(blockIdx.z,c,height+6,width+0) * mask_4d(blockIdx.x,c,6,0) + in_4d(blockIdx.z,c,height+6,width+1) * mask_4d(blockIdx.x,c,6,1) +
in_4d(blockIdx.z,c,height+6,width+2) * mask_4d(blockIdx.x,c,6,2) + in_4d(blockIdx.z,c,height+6,width+3) * mask_4d(blockIdx.x,c,6,3)+
in_4d(blockIdx.z,c,height+6,width+4) * mask_4d(blockIdx.x,c,6,4)+ in_4d(blockIdx.z,c,height+6,width+5) * mask_4d(blockIdx.x,c,6,5)+
in_4d(blockIdx.z,c,height+6,width+6) * mask_4d(blockIdx.x,c,6,6) ;

            var = var + in_4d(blockIdx.z,c,height+7,width+0) * mask_4d(blockIdx.x,c,7,0) + in_4d(blockIdx.z,c,height+7,width+1) * mask_4d(blockIdx.x,c,7,1) +
in_4d(blockIdx.z,c,height+7,width+2) * mask_4d(blockIdx.x,c,7,2) + in_4d(blockIdx.z,c,height+7,width+3) * mask_4d(blockIdx.x,c,7,3)+
in_4d(blockIdx.z,c,height+7,width+4) * mask_4d(blockIdx.x,c,7,4)+ in_4d(blockIdx.z,c,height+7,width+5) * mask_4d(blockIdx.x,c,7,5)+
in_4d(blockIdx.z,c,height+7,width+6) * mask_4d(blockIdx.x,c,7,6);



        }
        out_4d(blockIdx.z, blockIdx.x, height, width) = var; // storing the final results in out_4d
    }



    #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 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

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

    int out_size = ((Height_out*Width_out) * Map_out * Batch) * sizeof(float); // output size is batchsize * output channels * size of each output image
    int in_size = (Height*Width) * Channel * Batch * sizeof(float); // input size is input image dimensions * channels * batchsize
    int k_size = (K*K) * Map_out * Channel * sizeof(float); //each filter times input channels and output feature maps

    cudaMalloc((void**)device_input_ptr, in_size);
    cudaMalloc((void**)device_mask_ptr, k_size);
    cudaMalloc((void**)device_output_ptr, out_size);

    cudaMemcpy(*device_input_ptr, host_input, in_size, cudaMemcpyHostToDevice);
    cudaMemcpy(*device_mask_ptr, host_mask, k_size, cudaMemcpyHostToDevice);

}
```

```
__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 Channel, const int Height, const int Width, const int K)
{
    // Set the kernel dimensions and call the kernel
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;

    dim3 dimGrid(Map_out, ((Width_out + TILE_WIDTH -1)/TILE_WIDTH) * ((Height_out + TILE_WIDTH -1)/TILE_WIDTH), Batch);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

    conv_forward_kernel<<<dimGrid,dimBlock>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);

}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_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 out_size = (Height_out*Width_out) * Map_out * Batch * sizeof(float);

    // Copy the output back to host

    cudaMemcpy(host_output, device_output, out_size, cudaMemcpyDeviceToHost);

    // Free device memory

    cudaFree(device_output);
    cudaFree(device_input);
    cudaFree(device_mask);
}


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

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

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

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

    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

Using Streams to overlap computation with data transfer technique allows the CPU and GPU to collectively process the data and execute the computations parallelly. Since both CPU and GPU work together, the performance increases and reduces the overall execution time of the code. Furthermore, this technique is widely used in machine learning, computer graphics and other GPU accelerated applications.

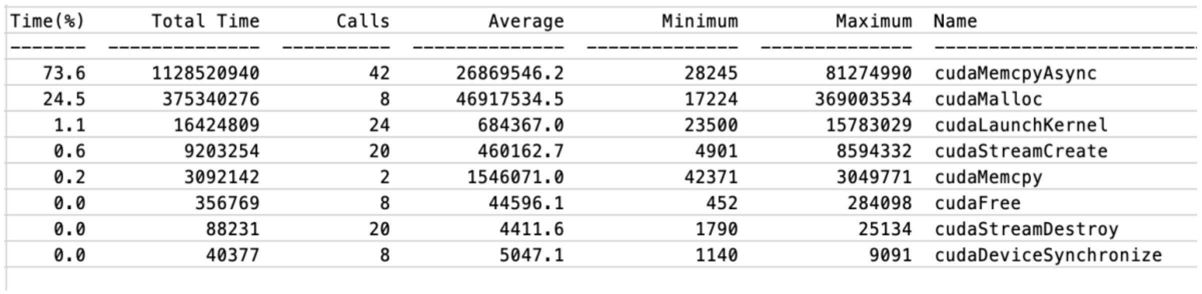
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

In this code, the Streaming technique is employed to increase the efficiency of data transfer and computation on the GPU. To achieve this, an array of CUDA streams is created to handle subsets of the input batch data. Each stream is then used to asynchronously copy data from the host to the device, launch the convolutional forward kernel, and copy the output data from the device to the host. By doing so, data transfer and computation can overlap, leading to improved performance. To ensure that all streams complete their operations before the function returns the output, the cudaDeviceSynchronize() function is called. This technique should increase the performance as we are shifting from serial data transfer to parallel and synchronizing it with computation. This technique has been implemented independently and it is a stand-alone procedure when compared to any previous 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).

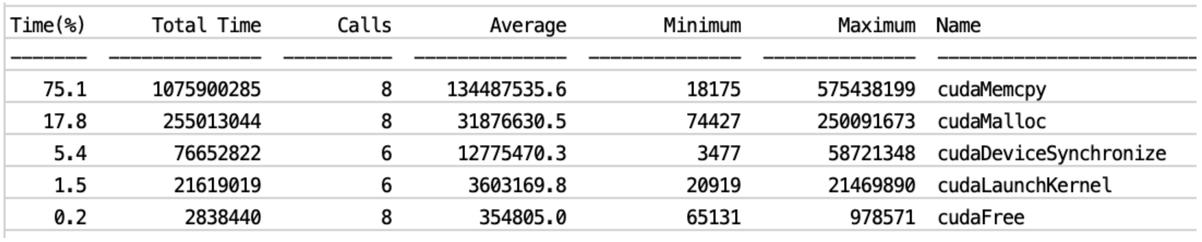
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.000631 ms*	*0.000667 ms*	*0m1.220s*	*0.86*
1000	*0.000974 ms*	*0.000758 ms*	*0m11.478s*	*0.886*
10000	*0.000817 ms*	*0.000814 ms*	*1m41.642s*	*0.8714*

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



```
Time(%)     Total Time       Calls         Average         Minimum         Maximum   Name
-------   --------------   ----------   --------------   --------------   --------------   -------------------------
   73.6       1128520940          42       26869546.2           28245         81274990   cudaMemcpyAsync
   24.5        375340276           8       46917534.5           17224        369003534   cudaMalloc
    1.1         16424809          24         684367.0           23500         15783029   cudaLaunchKernel
    0.6          9203254          20         460162.7            4901          8594332   cudaStreamCreate
    0.2          3092142           2        1546071.0           42371          3049771   cudaMemcpy
    0.0           356769           8          44596.1             452           284098   cudaFree
    0.0            88231          20           4411.6            1790            25134   cudaStreamDestroy
    0.0            40377           8           5047.1            1140             9091   cudaDeviceSynchronize
```

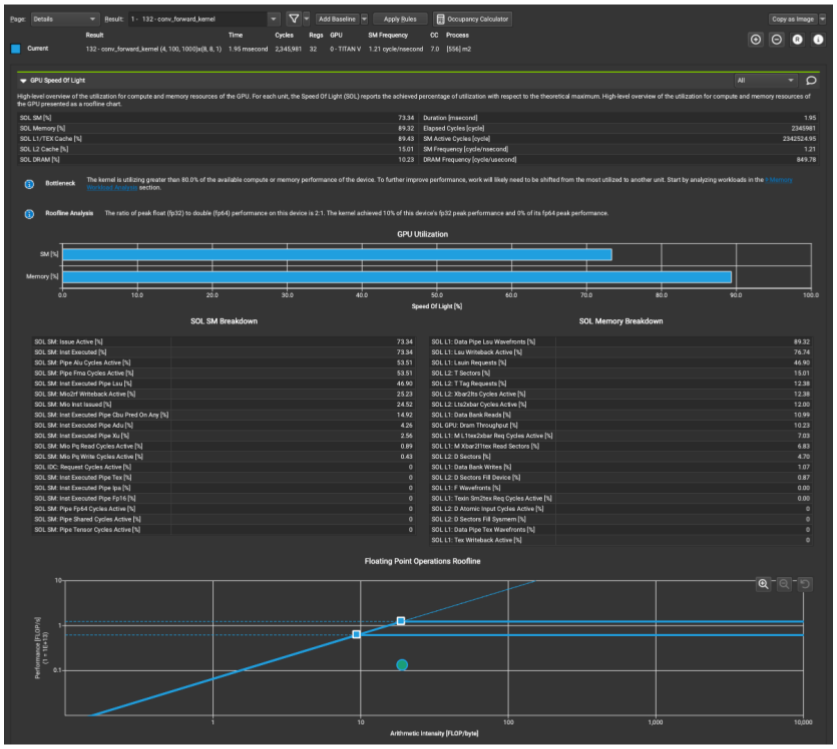
Fig 4a. CUDA API Statistics (nanoseconds) for Streaming technique for batch size 10k



```
Time(%)     Total Time       Calls         Average         Minimum         Maximum   Name
-------   --------------   ----------   --------------   --------------   --------------   -------------------------
   75.1       1075900285           8      134487535.6           18175        575438199   cudaMemcpy
   17.8        255013044           8       31876630.5           74427        250091673   cudaMalloc
    5.4         76652822           6       12775470.3            3477         58721348   cudaDeviceSynchronize
    1.5         21619019           6        3603169.8           20919         21469890   cudaLaunchKernel
    0.2          2838440           8         354805.0           65131           978571   cudaFree
```

Fig 4b. CUDA API Statistics (nanoseconds) for Baseline for batch size 10k

The Streaming technique has improved the performance with a drastic decrease in OP times. The accuracy for each batch size is comparable to the baseline model so is the total execution time. Furthermore, the CUDA API statistics shows that around 25% of the time has been spent on allocating memory in the streaming technique as compared to a similar time in baseline doing both allocating and synchronizing. It is because of the parallel loading and computation of data that such enhanced performance is achieved with streaming.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	132 - conv_forward_kernel (4, 100, 1000)x(8, 8, 1)	1.95 msecond	2,345,981	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[556] 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 [%]	73.34	Duration [msecond]	1.95
SOL Memory [%]	89.32	Elapsed Cycles [cycle]	2345981
SOL L1/TEX Cache [%]	89.43	SM Active Cycles [cycle]	2342524.95
SOL L2 Cache [%]	15.01	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	10.23	DRAM Frequency [cycle/usecond]	849.78

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 10% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

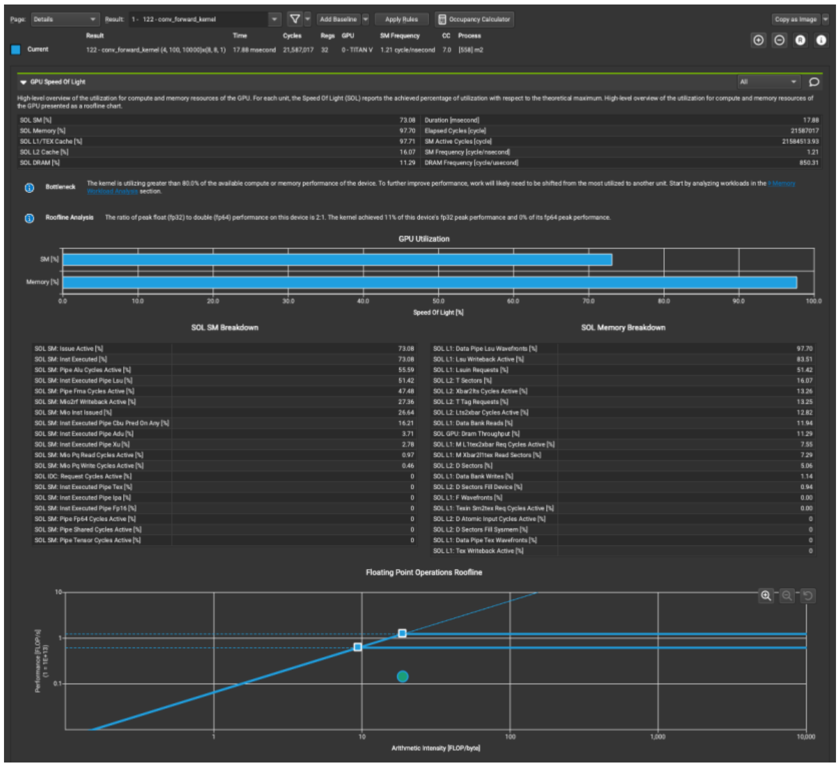
SOL SM: Issue Active [%]	73.34
SOL SM: Inst Executed [%]	73.34
SOL SM: Pipe Alu Cycles Active [%]	53.51
SOL SM: Pipe Fma Cycles Active [%]	53.51
SOL SM: Inst Executed Pipe Lsu [%]	46.90
SOL SM: Mio2rf Writeback Active [%]	25.23
SOL SM: Mio Inst Issued [%]	24.52
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	14.92
SOL SM: Inst Executed Pipe Adu [%]	4.26
SOL SM: Inst Executed Pipe Xu [%]	2.56
SOL SM: Mio Pq Read Cycles Active [%]	0.89
SOL SM: Mio Pq Write Cycles Active [%]	0.43
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	89.32
SOL L1: Lsu Writeback Active [%]	76.74
SOL L1: Lsuin Requests [%]	46.90
SOL L2: T Sectors [%]	15.01
SOL L2: T Tag Requests [%]	12.38
SOL L2: Xbar2lts Cycles Active [%]	12.38
SOL L2: Lts2xbar Cycles Active [%]	12.00
SOL L1: Data Bank Reads [%]	10.99
SOL GPU: Dram Throughput [%]	10.23
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.03
SOL L1: M Xbar2l1tex Read Sectors [%]	6.83
SOL L2: D Sectors [%]	4.70
SOL L1: Data Bank Writes [%]	1.07
SOL L2: D Sectors Fill Device [%]	0.87
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Floating Point Operations Roofline
10
1
0.1
Performance [FLOP/s] (1 = 1E+13)
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]

Fig 4c. Analysis of streaming technique using Nsight-Compute for batch size 10k



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	17.88 msecond	21,587,017	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[558] m2

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 [%]	73.08	Duration [msecond]	17.88
SOL Memory [%]	97.70	Elapsed Cycles [cycle]	21587017
SOL L1/TEX Cache [%]	97.71	SM Active Cycles [cycle]	21584513.93
SOL L2 Cache [%]	16.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.29	DRAM Frequency [cycle/usecond]	850.31

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	73.08
SOL SM: Inst Executed [%]	73.08
SOL SM: Pipe Alu Cycles Active [%]	55.59
SOL SM: Inst Executed Pipe Lsu [%]	51.42
SOL SM: Pipe Fma Cycles Active [%]	47.48
SOL SM: Mio2rf Writeback Active [%]	27.36
SOL SM: Mio Inst Issued [%]	26.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.21
SOL SM: Inst Executed Pipe Adu [%]	3.71
SOL SM: Inst Executed Pipe Xu [%]	2.78
SOL SM: Mio Pq Read Cycles Active [%]	0.97
SOL SM: Mio Pq Write Cycles Active [%]	0.46
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	97.70
SOL L1: Lsu Writeback Active [%]	83.51
SOL L1: Lsuin Requests [%]	51.42
SOL L2: T Sectors [%]	16.07
SOL L2: Xbar2lts Cycles Active [%]	13.26
SOL L2: T Tag Requests [%]	13.25
SOL L2: Lts2xbar Cycles Active [%]	12.82
SOL L1: Data Bank Reads [%]	11.94
SOL GPU: Dram Throughput [%]	11.29
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.55
SOL L1: M Xbar2l1tex Read Sectors [%]	7.29
SOL L2: D Sectors [%]	5.06
SOL L1: Data Bank Writes [%]	1.14
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 4d. Analysis of baseline technique using Nsight-Compute for batch size 10k

With a similar arithmetic intensity [FLOP/byte] and consumption of SM, there is a slight decrease of 8% in the memory utilization in the newly implemented streaming technique. Overall, the streaming technique does help in achieving better performance than the baseline.

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

1. *https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/*
2. *https://people.cs.vt.edu/yongcao/teaching/cs5234/spring2013/slides/Lecture9.pdf*
3. *https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Task-Parallelism-Using-Different-Kernels*
4. *https://www.cs.utexas.edu/~pingali/CS380C/2020/project4/H2DUnifiedMemory.pdf*

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"

#define TILE_WIDTH 8

__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;
    // (void)Height_out; // silence declared but never referenced warning. remove this line when you start working
    // (void)Width_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

    int W_size = (Width_out + TILE_WIDTH - 1) / 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) mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
```

```
    // Insert your GPU convolution kernel code here

    // int H_size = Height_out/TILE_WIDTH;
    int y = blockIdx.y;
    int H = (y / W_size) * TILE_WIDTH + threadIdx.y;
    int W = (y % W_size) * TILE_WIDTH + threadIdx.x;

    if (H < Height_out && W < Width_out){ // for all batches, for all height and width pixel values
        float inter = 0.0f; // declaring a temp variable
        for (int C = 0; C < Channel; C++) { // sum over all channels
            for (int k = 0; k < K; k++){ // loop over KxK filter
                for (int i = 0; i < K; i++){
                    inter += in_4d(blockIdx.z, C, H + k, W + i) * mask_4d(blockIdx.x, C, k, i); // calculating convolution and adding the
intermediate results to inter variable
                }
            }
        }
        out_4d(blockIdx.z, blockIdx.x, H, W) = inter; // storing the final results in out_4d
    }

#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 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

#define stream_size 10

    // Allocate memory and copy over the relevant data structures to the GPU

    const int Height_out = Height - K + 1; // Calculate output height and width based on input height, width, and filter size.
    const int Weight_out = Width - K + 1;

    int W_size = (Weight_out + TILE_WIDTH - 1) / TILE_WIDTH; // Determine number of thread blocks needed for each dimension based
on output height, width, and tile size.
    int H_size = (Height_out + TILE_WIDTH - 1) / TILE_WIDTH;

    float* host_output_temp = (float*)host_output; // Cast the host_output pointer to a float pointer for convenience.
```

```
  int input_batch_size = (Batch * Channel * Height * Width) / stream_size; // Calculate the input and output batch sizes per stream.
  int output_batch_size = (Batch * Map_out * Height_out * Weight_out) / stream_size;

  dim3 gridDim(Map_out, W_size * H_size, Batch/stream_size); // Set the dimensions of the CUDA kernel grid and thread blocks.
  dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);

  cudaStream_t A[stream_size]; // Create an array of CUDA streams.
  for (int x = 0; x < stream_size; x++){
    cudaStreamCreate(&A[x]); // Create a CUDA stream for each index in the array.
  }



  int out_size = ((Height_out*Weight_out) * Map_out * Batch) * sizeof(float); // output size is batchsize * output channels * size of each
output image
  int in_size = (Height*Width) * Channel * Batch * sizeof(float); // input size is input image dimensions * channels * batchsize
  int k_size = (K*K) * Map_out * Channel * sizeof(float); //each filter times input channels and output feature maps

  cudaMalloc((void**)device_input_ptr, in_size); // Allocate memory on the device for input, mask, and output data.
  cudaMalloc((void**)device_mask_ptr, k_size);
  cudaMalloc((void**)device_output_ptr, out_size);

  cudaMemcpyAsync(*device_mask_ptr, host_mask, k_size, cudaMemcpyHostToDevice, A[0]); // Copy the filter mask from the host to
the device asynchronously using stream 0.

  for (int i = 0; i < stream_size; i++){ // Loop over each stream to perform convolution on a batch subset of the input data.
    int input_offset = input_batch_size * i; // Calculate the input and output batch offsets for the current stream.
    int output_offset = output_batch_size * i;
    // Copy the input batch data from the host to the device asynchronously for the current stream.
    cudaMemcpyAsync((*device_input_ptr) + input_offset, host_input + input_offset, input_batch_size * sizeof(float),
cudaMemcpyHostToDevice, A[i]);
    // Launch the convolution forward kernel for the current stream
    conv_forward_kernel<<<gridDim, blockDim, 0, A[i]>>>((*device_output_ptr) + output_offset, (*device_input_ptr) + input_offset,
*device_mask_ptr, Batch, Map_out,Channel, Height, Width, K);
    // Copy the output batch data from the device to the host asynchronously for the current stream.
    cudaMemcpyAsync(host_output_temp + output_offset, (*device_output_ptr) + output_offset, output_batch_size * sizeof(float),
cudaMemcpyDeviceToHost, A[i]);
  }
  cudaDeviceSynchronize(); // Wait for all streams to complete their operations before proceeding.



  for (int x = 0; x < stream_size; x++)
    cudaStreamDestroy(A[x]); // Release allocated resources.
  cudaFree(device_input_ptr);
  cudaFree(device_mask_ptr);
  cudaFree(device_output_ptr);

#undef STREAM_NUM

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

}


__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 Channel, const int Height, const int Width, const int K)
{
  // // Set the kernel dimensions and call the kernel
  // const int Height_out = Height - K + 1;
  // const int Width_out = Width - K + 1;

  // int w_size = ceil((1.0*Width_out)/TILE_WIDTH);
  // int h_size = ceil((1.0*Height_out)/TILE_WIDTH);

  // dim3 dimGrid(Map_out, w_size * h_size, ceil((1.0*Batch)/TILE_WIDTH));
  // dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, TILE_WIDTH);
  // conv_forward_kernel<<<dimGrid,dimBlock>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width,
K);

  return;

}


__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_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 out_size = (Height_out*Width_out) * Map_out * Batch * sizeof(float);
  // // Copy the output back to host

  // cudaMemcpy(host_output, device_output, out_size, cudaMemcpyDeviceToHost);

  // // Free device memory

  // cudaFree(device_output);
  // cudaFree(device_input);
  // cudaFree(device_mask);
  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;
  }
}
```

5. **Optimization 5: Input channel reduction: tree**

   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

The input channel reduction – tree technique takes advantage of reduces the number of channels and processes the data at once. Therefore, it helps in speeding up the training and inference of CNNs by reducing the memory access and computations. This technique is widely used in computer vision, object detection and other real-world detection applications. Hence, implementing this technique will support in obtaining the objective as well as assist in keeping up to date with the modern approaches.

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 input channel reduction: tree technique is used to reduce computational cost associated with CNNs. It removes any redundant channels from the input without compromising on the accuracy. Basically, this technique removes the least important channels from the input feature maps and passes the rest to the next layer. This step is repeated until the desired compression is not achieved. The performance of the forward convolution should increase as there are lesser computations, and the process is faster. This technique has been implemented independently and it is a stand-alone procedure when compared to any previous 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	*0.161987 ms*	*0.820682 ms*	*0m1.263s*	*0.86*
1000	*1.60816 ms*	*7.99982 ms*	*0m10.876s*	*0.886*
10000	*15.8403 ms*	*87.4281 ms*	*1m40.736s*	*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

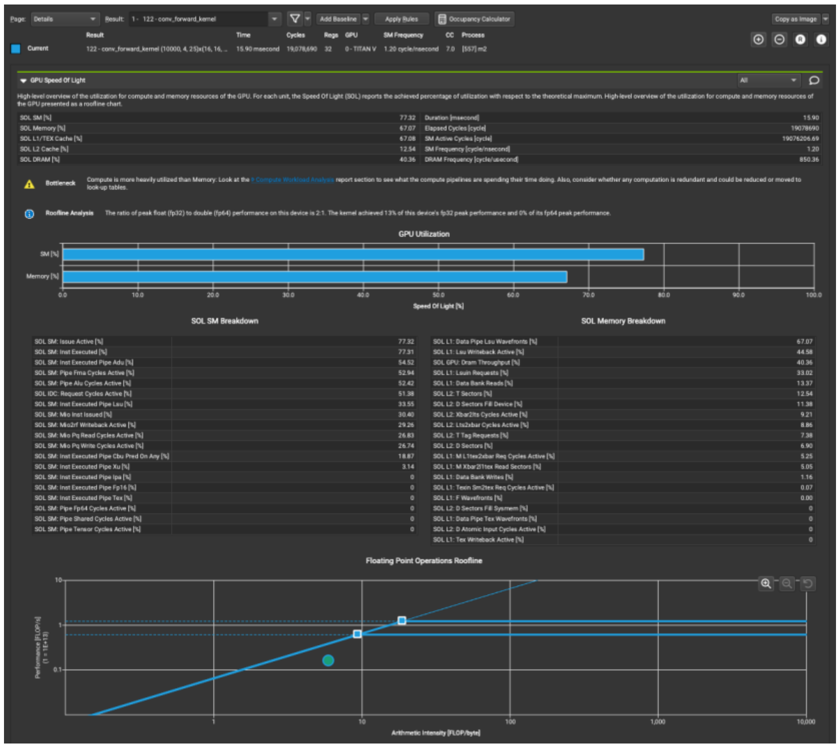
```
Time(%)      Total Time      Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   78.4      1081829051           6    180304841.8           18176       583688796  cudaMemcpy
   12.7       175200829           6     29200138.2          312343       171970199  cudaMalloc
    7.5       104097176           6     17349529.3            3202        88177923  cudaDeviceSynchronize
    1.2        16773294           6      2795549.0           16309        16657579  cudaLaunchKernel
    0.2         2415689           6       402614.8           86014          906788  cudaFree
    0.0          347106           2       173553.0          172304          174802  cudaMemcpyToSymbol
```

Fig 5a. CUDA API Statistics (nanoseconds) for input channel reduction: tree technique for batch size 10k

```
Time(%)      Total Time      Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   75.1      1075900285           8    134487535.6           18175       575438199  cudaMemcpy
   17.8       255013044           8     31876630.5           74427       250091673  cudaMalloc
    5.4        76652822           6     12775470.3            3477        58721348  cudaDeviceSynchronize
    1.5        21619019           6      3603169.8           20919        21469890  cudaLaunchKernel
    0.2         2838440           8       354805.0           65131          978571  cudaFree
```

Fig 5b. CUDA API Statistics (nanoseconds) for Baseline for batch size 10k

The input channel reduction: tree technique has a similar total execution time as the baseline code. Furthermore, the OP times are also similar with a slightly slower execution in layer 2. This variation might be because of more time that was spent in copying data to and from memory and synchronizing the devices. However, the time spent after allocating the memory is lesser in the input channel reduction: tree technique over baseline which happens due to discounting least important channels.



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (10000, 4, 25)x(16, 16, ...	15.90 msecond	19,078,690	32	0 - TITAN V	1.20 cycle/nsecond	7.0	[557] m2

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 [%]	77.32	Duration [msecond]	15.90
SOL Memory [%]	67.07	Elapsed Cycles [cycle]	19078690
SOL L1/TEX Cache [%]	67.08	SM Active Cycles [cycle]	19076206.69
SOL L2 Cache [%]	12.54	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	40.36	DRAM Frequency [cycle/usecond]	850.36

Bottleneck Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 13% of this device's fp32 peak performance and 0% of its fp64 peak performance.

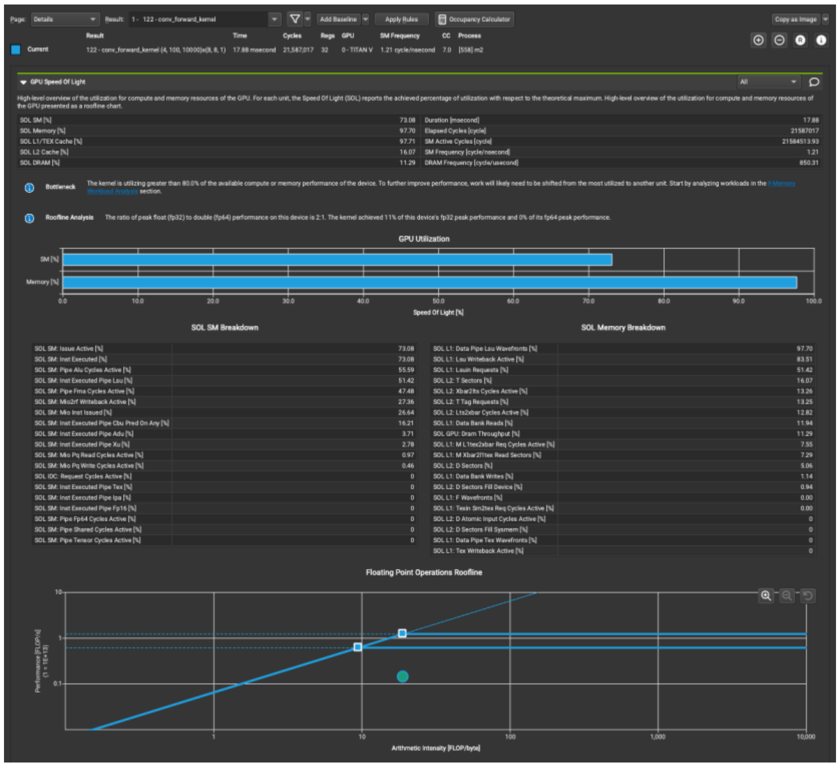
SOL SM Breakdown

SOL SM: Issue Active [%]	77.32
SOL SM: Inst Executed [%]	77.31
SOL SM: Inst Executed Pipe Adu [%]	54.52
SOL SM: Pipe Fma Cycles Active [%]	52.94
SOL SM: Pipe Alu Cycles Active [%]	52.42
SOL IDC: Request Cycles Active [%]	51.38
SOL SM: Inst Executed Pipe Lsu [%]	33.55
SOL SM: Mio Inst Issued [%]	30.40
SOL SM: Mio2rf Writeback Active [%]	29.26
SOL SM: Mio Pq Read Cycles Active [%]	26.83
SOL SM: Mio Pq Write Cycles Active [%]	26.74
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	18.87
SOL SM: Inst Executed Pipe Xu [%]	3.14
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	67.07
SOL L1: Lsu Writeback Active [%]	44.58
SOL GPU: Dram Throughput [%]	40.36
SOL L1: Lsuin Requests [%]	33.02
SOL L1: Data Bank Reads [%]	13.37
SOL L2: T Sectors [%]	12.54
SOL L2: D Sectors Fill Device [%]	11.38
SOL L2: Xbar2lts Cycles Active [%]	9.21
SOL L2: Lts2xbar Cycles Active [%]	8.86
SOL L2: T Tag Requests [%]	7.38
SOL L2: D Sectors [%]	6.90
SOL L1: M L1tex2xbar Req Cycles Active [%]	5.25
SOL L1: M Xbar2l1tex Read Sectors [%]	5.05
SOL L1: Data Bank Writes [%]	1.16
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.07
SOL L1: F Wavefronts [%]	0.00
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L1: Tex Writeback Active [%]	0

Fig 5c. Analysis of input channel reduction: tree technique using Nsight-Compute for batch size 10k



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 100, 10000)x(8, 8, 1)	17.88 msecond	21,587,017	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[558] 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 [%]	73.08	Duration [msecond]	17.88
SOL Memory [%]	97.70	Elapsed Cycles [cycle]	21587017
SOL L1/TEX Cache [%]	97.71	SM Active Cycles [cycle]	21584513.93
SOL L2 Cache [%]	16.07	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	11.29	DRAM Frequency [cycle/usecond]	850.31

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	73.08
SOL SM: Inst Executed [%]	73.08
SOL SM: Pipe Alu Cycles Active [%]	55.59
SOL SM: Inst Executed Pipe Lsu [%]	51.42
SOL SM: Pipe Fma Cycles Active [%]	47.48
SOL SM: Mio2rf Writeback Active [%]	27.36
SOL SM: Mio Inst Issued [%]	26.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	16.21
SOL SM: Inst Executed Pipe Adu [%]	3.71
SOL SM: Inst Executed Pipe Xu [%]	2.78
SOL SM: Mio Pq Read Cycles Active [%]	0.97
SOL SM: Mio Pq Write Cycles Active [%]	0.46
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	97.70
SOL L1: Lsu Writeback Active [%]	83.51
SOL L1: Lsuin Requests [%]	51.42
SOL L2: T Sectors [%]	16.07
SOL L2: Xbar2lts Cycles Active [%]	13.26
SOL L2: T Tag Requests [%]	13.25
SOL L2: Lts2xbar Cycles Active [%]	12.82
SOL L1: Data Bank Reads [%]	11.94
SOL GPU: Dram Throughput [%]	11.29
SOL L1: M L1tex2xbar Req Cycles Active [%]	7.55
SOL L1: M Xbar2l1tex Read Sectors [%]	7.29
SOL L2: D Sectors [%]	5.06
SOL L1: Data Bank Writes [%]	1.14
SOL L2: D Sectors Fill Device [%]	0.94
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

Floating Point Operations Roofline
10
1
0.1
Performance [FLOP/s] (1 = 1E+13)
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]

Fig 5d. Analysis of baseline technique using Nsight-Compute for batch size 10k

With a drastic drop in arithmetic intensity [FLOP/byte] and memory utilization, the newly implemented streaming technique provides better performance compared to baseline. Overall, the input channel reduction: tree technique does help in achieving better performance than the baseline.

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

1. *http://lumetta.web.engr.illinois.edu/408-S20/slide-copies/ece408-lecture15-S20.pdf*
2. *https://core.ac.uk/download/pdf/147691424.pdf*

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"

#define TILE_WIDTH 16

__constant__ float constant_mem_obj[8000];

__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:
    y - output
    x - input
    k - 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)
    */

    extern __shared__ float shared_mem_obj[];
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;

   // (void)Height_out; // silence declared but never referenced warning. remove this line when you start working
    // (void)Width_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) * (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) constant_mem_obj[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
    #define tree(i2, i1, i0) shared_mem_obj[i2 * TILE_WIDTH * Channel + i1 * Channel + i0]
    // Insert your GPU convolution kernel code here

    int blockx = blockIdx.x;
    int blocky = blockIdx.y;
    // int blockz = blockIdx.z;
    int threadx = threadIdx.x;
    int thready = threadIdx.y;
    int threadz = threadIdx.z;

    // int numblock_eachcolumn = (Width_out - 1)/TILE_WIDTH + 1;
    int block_count = (Width_out - 1)/TILE_WIDTH + 1;
    int output_width = TILE_WIDTH * (blockIdx.z % block_count) + threadIdx.x;
    int output_height = TILE_WIDTH * (blockIdx.z/block_count) + threadIdx.y;

    if (output_height < Height_out && output_width < Width_out)
    {
        float inter = 0;

        for (int p = 0; p < K; p++)
        {
            for (int q = 0; q < K; q++)
            {
                inter += in_4d(blockx, threadz, output_height + p, output_width + q) * mask_4d(blocky, threadz, p, q);
            }
        }
        tree(thready, threadx, threadz) = inter;



        // tree reduction
        for (int str = 1; str < Channel; str *= 2)
        {
            __syncthreads();
            if ((threadz%(2*str) == 0) && (threadz + str < Channel))
                tree(thready, threadx, threadz) += tree(thready, threadx, threadz + str);
        }
        __syncthreads();
        if (threadz == 0)
            out_4d(blockx, blocky, output_height, output_width) = tree(thready, threadx, 0);

    }

#undef out_4d
#undef in_4d
```

```
#undef mask_4d
#undef tree
}



__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

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

    int out_size = ((Height_out*Width_out) * Map_out * Batch) * sizeof(float); // output size is batchsize * output channels * size of each
output image
    int in_size = (Height*Width) * Channel * Batch * sizeof(float); // input size is input image dimensions * channels * batchsize
    int k_size = (K*K) * Map_out * Channel * sizeof(float); //each filter times input channels and output feature maps

    cudaMalloc((void**)device_input_ptr, in_size);

    cudaMalloc((void**)device_output_ptr, out_size);

    cudaMemcpy(*device_input_ptr, host_input, in_size, cudaMemcpyHostToDevice);

    cudaMemcpyToSymbol(constant_mem_obj, host_mask, k_size);

}


__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 Channel, const int Height, const int Width, const int K)
{
    // Set the kernel dimensions and call the kernel

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

    dim3 dimGrid(Batch, Map_out, ceil((1.0 * Height_out)/TILE_WIDTH)*ceil((1.0 * Width_out)/TILE_WIDTH));
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, Channel);
    conv_forward_kernel<<<dimGrid, dimBlock, TILE_WIDTH * TILE_WIDTH * Channel * sizeof(float)>>>(device_output, device_input,
device_mask, Batch, Map_out, Channel, Height, Width, K);
}
```

```
__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_mask,
const int Batch, const int Map_out, const int Channel, const int Height, const int Width, const int K)
{
    // Copy the output back to host
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;

    int out_size = (Height_out*Width_out) * Map_out * Batch * sizeof(float);
    cudaMemcpy(host_output, device_output, out_size, cudaMemcpyDeviceToHost);

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


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

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

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