ECE 408 FA19 Project Report

Team: rtx5700ti
Members: Adit Umakanth, Advaith Ravikumar, Murugan Narayanan
NetID: adityau2, advaith2, mrn2
UIN: 674909598, 677014085, 670975133
School Affiliation: uiuc (on campus students)

Table of Contents

MILESTONE 1 & 2 Report

List of kernels that collectively consume more than 90% of the program time

GPU activities:
31.02% 34.114ms 20 1.7057ms 1.0880us 31.926ms [CUDA memcpy HtoD]

18.02% 19.822ms 1 19.822ms 19.822ms 19.822ms volta_scudnn_128x64_relu_interior_nn_v1

17.39% 19.123ms 4 4.7806ms 4.7767ms 4.7852ms volta_gcgemm_64x32_nt

8.71% 9.5779ms 4 2.3945ms 2.0010ms 3.1155ms void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=0, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)

7.91% 8.7044ms 1 8.7044ms 8.7044ms 8.7044ms volta_sgemm_128x128_tn

6.61% 7.2648ms 2 3.6324ms 25.504us 7.2392ms voi op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)

5.82% 6.3969ms 4 1.5992ms 1.2706ms 2.0309ms void fft2d_r2c_32x32<float, bool=0, unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)

3.96% 4.3558ms 1 4.3558ms 4.3558ms 4.3558ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

List: [CUDA memcpy HtoD], volta_scudnn_128x64_relu_interior_nn_v1, volta_gcgemm_64x32_nt, fft2d_c2r_32x32, volta_sgemm_128x128_tn, op_generic_tensor_kernel, fft2d_r2c_32x32, cudnn::detail::pooling_fw_4d_kernel

List of all CUDA API calls that collectively consume more than 90% of the program time
API calls:
44.10% 3.35356s 22 152.43ms 14.352us 1.71565s cudaStreamCreateWithFlags

30.82% 2.34363s 24 97.651ms 75.059us 2.33580s cudaMemGetInfo

21.80% 1.65752s 19 87.238ms 1.2160us 442.33ms cudaFree

List: cudaStreamCreateWithFlags, cudaMemGetInfo, cudaFree

Difference Between Kernels and API Calls

The kernel is a function compiled and intended to be executed on a computing unit different from the CPU, such as a GPU. Once the kernel is called from the host, it is executed entirely on the GPU to completion. The results of any computation done has to be copied from/to the GPU memory into/from the CPU memory.
API calls are functions called by the host to interact with the GPU. This includes things like copying and allocating memory to the GPU or getting information about the GPU. An example of the API call is cudaMalloc or cudaFree.

CPU OUTPUT

✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

CPU RUNTIME

19.36user 7.12system 0:10.17elapsed 260%CPU (0avgtext+0avgdata 6044932maxresident)k
0inputs+2824outputs (0major+1598360mino
r)pagefaults 0swaps

GPU OUTPUT

✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

GPU RUNTIME

4.99user 3.18system 0:04.81elapsed 169%CPU (0avgtext+0avgdata 2973968ma
xresident)k
0inputs+1712outputs (0major+730591minor)pagefaults 0swaps

CPU Convolution Program Run Time

87.36user 11.75system 1:15.18elapsed 131%CPU (0avgtext+0avgdata 6042528maxresident)k
0inputs+0outputs (0major+2305263minor)pagefaults 0swaps

List of Op Times

Op Time: 10.779369
Op Time: 59.818806
Correctness: 0.7653 Model: ece408

MILESTONE 3 Report

TIMING OUTPUT

✱ Running /usr/bin/time python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000406
Op Time: 0.003461
Correctness: 0.76 Model: ece408
5.03user 2.84system 0:07.16elapsed 109%CPU (0avgtext+0avgdata 2783908maxresident)k
0inputs+4568outputs
(0major+634601minor)pagefaults 0swaps

✱ Running /usr/bin/time python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.004505
Op Time: 0.034755
Correctness: 0.767 Model: ece408
5.07user 3.06system 0:04.62elapsed 175%CPU (0avgtext+0avgdata 2806940maxresident)k
0inputs+0outputs (0major+640458minor)pagefaults 0swaps

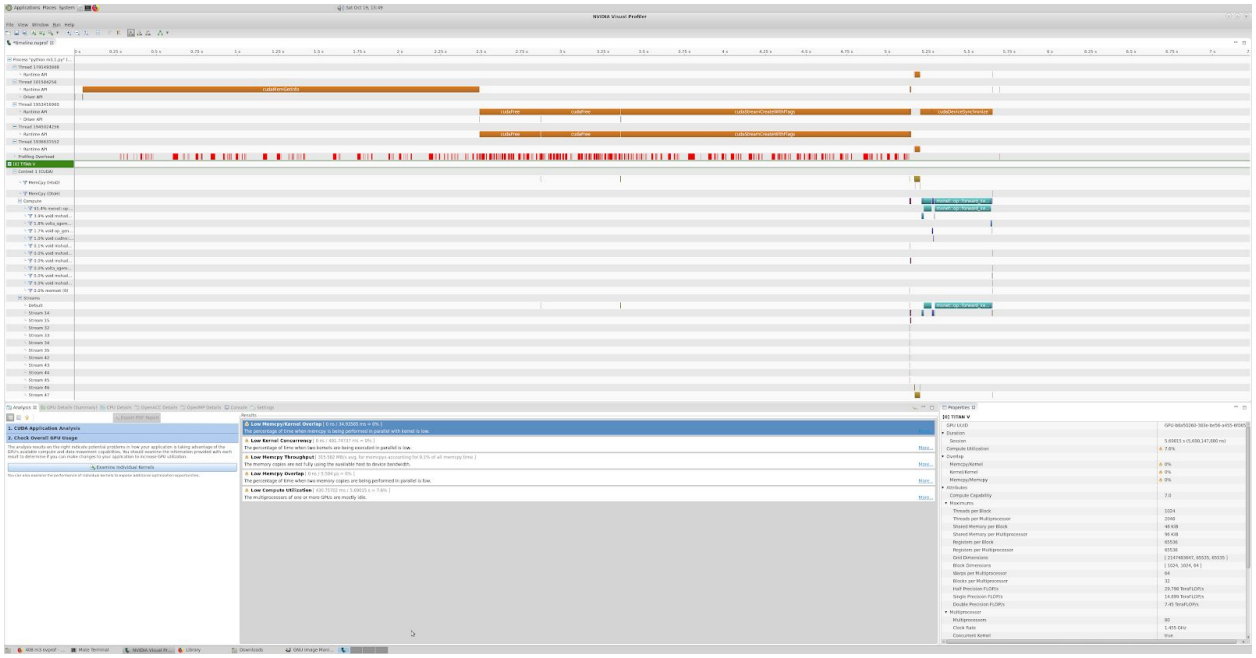
✱ Running /usr/bin/time python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.051133
Op Time: 0.326978
Correctness: 0.7653 Model: ece408
5.30user 2.66system 0:05.04elapsed 157%CPU (0avgtext+0avgdata 2974616maxresident)k
0inputs+0outputs (0major+729886minor)pagefaults 0swaps

NVPROF OUTPUT

✱ Running /usr/bin/time python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.051133
Op Time: 0.326978

```
Correctness: 0.7653 Model: ece408
5.30user 2.66system 0:05.04elapsed 157%CPU (0avgtext+0avgdata 2974616maxresident)k
0inputs+0outputs (0major+729886minor)pagefaults 0swaps
✱ Running nvprof python m3.1.py
Loading fashion-mnist data... done
==528== NVPROF is profiling process 528, command: python m3.1.py
Loading model... done
New Inference
Op Time: 0.048657
Op Time: 0.324925
Correctness: 0.7653 Model: ece408
==528== Profiling application: python m3.1.py
==528== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:
83.39%  373.52ms         2  186.76ms  48.623ms  324.89ms  mxnet::op::forward_kernel(float*,
float const *, float const *, int, int, int, int, int, int)
8.37%  37.510ms        20  1.8755ms  1.0560us  35.118ms  [CUDA memcpy HtoD]
3.73%  16.726ms         2  8.3632ms  3.0561ms  13.670ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
List: forward_kernel, CUDA memcpy HostToDevice, MapPlanLargeKernel


API calls:
40.35%  3.24361s        22  147.44ms  15.033us  1.69584s  cudaStreamCreateWithFlags
30.29%  2.43526s        22  110.69ms  72.338us  2.42973s  cudaMemGetInfo
21.12%  1.69759s        18  94.311ms  1.2300us  461.39ms  cudaFree
 4.85%  390.27ms         6  65.046ms  3.2600us  324.90ms  cudaDeviceSynchronize

List: cudaStreamCreateWithFlags, cudaMemGetInfo, cudaFree, cudaDeviceSynchronize
```

NVVP Timeline:



The kernel we wrote only accounts for 345 ms out of the total runtime of about 5.6 s. It was interesting to see that the majority of the execution time is not our kernel, but rather cudaStreamCreateWithFlags and cudaMemGetInfo.

MILESTONE 4 REPORT

4.1 Introduction

For this milestone, we chose to tackle 3 of the optimizations in the suggested list on the project GitHub page. We will explain these optimizations and show the effect of said optimizations on the runtime of the neural network.

It is important to note that all our optimizations output the correct accuracy as mentioned on the project GitHub page, so this means that there is no change to the output of each convolution operation.

As recommended, we implemented these optimizations independently to better showcase the effect that each one has on the runtime of the convolution. The optimizations we chose to implement are as follows:

- Shared Memory Convolution
- Weight Matrix (Kernel Values) in Constant Memory
- Multiple Kernel Implementations for Different Layer Sizes

4.2 Baseline

Our baseline kernel without any optimizations is a bit different from our kernel from Milestone 3. We are now using the threads in the *z* dimension to process different images in parallel, and each thread is looping through the image channels instead. We were doing the opposite in Milestone 3. This change has significantly improved our execution time.

4.2.1 Timing Output

Op Time: 0.058203
Op Time: 0.140968
Correctness: 0.7653 Model: ece408

4.2.2 NVPROF Output

==353== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 73.59% 199.12ms 2 99.561ms 58.172ms 140.95ms mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
12.64% 34.199ms 20 1.7099ms 1.0560us 32.092ms [CUDA memcpy HtoD]
6.28% 16.983ms 2 8.4917ms 3.0878ms 13.896ms void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,

mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul, mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
2.93% 7.9166ms 1 7.9166ms 7.9166ms 7.9166ms volta_sgemm_128x128_tn
2.71% 7.3307ms 2 3.6653ms 25.312us 7.3054ms void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
1.63% 4.4010ms 1 4.4010ms 4.4010ms 4.4010ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

API calls: 43.30% 3.19750s 22 145.34ms 29.914us 1.59818s cudaStreamCreateWithFlags
30.67% 2.26454s 22 102.93ms 69.659us 2.25993s cudaMemGetInfo
21.54% 1.59030s 18 88.350ms 1.4450us 417.40ms cudaFree
2.93% 216.12ms 6 36.021ms 2.6080us 140.96ms cudaDeviceSynchronize
0.94% 69.293ms 9 7.6993ms 26.442us 32.284ms cudaMemcpy2DAsync
0.27% 19.702ms 29 679.37us 2.1270us 11.120ms cudaStreamSynchronize
0.17% 12.268ms 66 185.88us 9.0930us 4.5967ms cudaMalloc
0.07% 5.4763ms 4 1.3691ms 867.92us 1.8577ms cudaGetDeviceProperties
0.03% 2.4806ms 375 6.6140us 384ns 334.65us cuDeviceGetAttribute
0.02% 1.3124ms 2 656.21us 643.51us 668.91us cudaHostAlloc
0.02% 1.3044ms 912 1.4300us 468ns 19.038us cudaFuncSetAttribute

4.2.3 NVVP Timeline

4.2.4 Relevant Code

Kernel code:

```
for (int m = 0; m < M; m++){     // Loop through images
      conv_val = 0.0; // Reset val for new image
      for (int c = 0; c < C; c++){     // Loop through input features
            // Loop through 2D convolution kernel/filter
            for (int kx = 0; kx < K; kx++){
                  for (int ky = 0; ky < K; ky++){
                        if ((tz < B) && (ty + ky < H) && (tx + kx <
W)){ // if dimensions are within input bounds and M
                        conv_val += k4d(m, c, ky, kx) * x4d(tz, c,
ty+ky, tx+kx);    // do convolution
                        }
                  }
            }
      }
```

4.3 Shared Memory Convolution

The main bottleneck with a lot of GPU algorithms is the bandwidth on global memory. Constantly writing and reading data to and from the global memory slows down the access time for each thread and can cut down the theoretical speed of algorithms by orders of magnitude.

Using shared memory is one way to get around this bottleneck when there is a high level of data reuse, like in convolution. Each thread block has its own shared memory. Data is copied here

from global memory, but is only done so once, at the start of the kernel function. Then for subsequent reads, data can be fetched from the shared memory instead of from global memory.

This is advantageous for multiple reasons. Each thread block only accesses its own shared memory, so there are no bandwidth issues unlike when all thread blocks in the kernel try to read and write to global memory. The second advantage is that shared memory is much faster than global memory, even without taking bandwidth issues into consideration. According to the Nvidia Developer Blog, “Because it is on-chip, shared memory is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency “.

Similar to MP4, we used strategy 2 to load the input image snippet that was going to be used in the convolution. We looped through input channels and got the (X,Y, Image) position from the thread indices to load the correct input addresses. Similar to the MP, we load C*(TILE_WIDTH + K - 1)^2 input elements, where C is the number of input channels, and K is the convolution filter size. Each element is used M times in our convolution as we loop through the output channels.

4.3.1 Timing Output

Op Time: 0.030706
Op Time: 0.103325
Correctness: 0.7653 Model: ece408

4.3.2 NVPROF Output

==352== Profiling result:

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.36%  133.95ms         2  66.976ms  30.655ms  103.30ms
mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                  16.25%  32.797ms        20  1.6398ms  1.0240us  30.606ms  [CUDA memcpy
HtoD]
                   7.34%  14.819ms         2  7.4093ms  2.9247ms  11.894ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                   4.01%  8.1008ms         1  8.1008ms  8.1008ms  8.1008ms
volta_sgemm_128x128_tn
                   3.57%  7.2106ms         2  3.6053ms  25.216us  7.1854ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
```

2.17% 4.3736ms 1 4.3736ms 4.3736ms 4.3736ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

API calls: 42.98% 3.12682s 22 142.13ms 14.367us 1.60838s cudaStreamCreateWithFlags
30.77% 2.23869s 22 101.76ms 68.516us 2.23426s cudaMemGetInfo
21.41% 1.55756s 18 86.531ms 1.1890us 418.63ms cudaFree
2.05% 148.79ms 6 24.799ms 2.8080us 103.30ms cudaDeviceSynchronize
0.91% 66.412ms 9 7.3791ms 29.860us 30.750ms cudaMemcpy2DAsync
0.79% 57.600ms 66 872.72us 6.2700us 32.962ms cudaMalloc
0.48% 34.815ms 912 38.174us 444ns 8.1835ms cudaFuncSetAttribute
0.27% 19.741ms 29 680.73us 2.2350us 10.992ms cudaStreamSynchronize
0.13% 9.3560ms 6 1.5593ms 1.8700us 9.3257ms cudaEventCreate
0.07% 4.9837ms 4 1.2459ms 436.31us 1.8680ms cudaGetDeviceProperties
0.03% 2.4735ms 375 6.5950us 407ns 332.83us cuDeviceGetAttribute
0.02% 1.4020ms 216 6.4900us 1.2480us 162.43us cudaEventCreateWithFlags

4.3.3 NVVP Timeline

4.3.4 Relevant Code

Kernel code:

```
__shared__ float shared_mem[MAX_C*SM_DIM*SM_DIM];
…
// copy shared memory
     // Images * In_Channels * Y * X;

     for (int c = 0; c < C; c++){
     int index = (c*SM_DIM*SM_DIM) + (ty*SM_DIM) + (tx);
     // if (loc_x_i==0 && loc_y_i==0){
     //   printf("IDX: %d, C: %d %d, tzyx = %d %d %d, X:%d, Y:%d Z:
%d\n", index, c, C, tz, ty, tx, loc_x_i, loc_y_i, loc_z_i);
     // }
     // __syncthreads();
     if (loc_z_i < B && loc_y_i < H && loc_x_i < W){
          shared_mem[index] = x4d(loc_z_i, c, loc_y_i, loc_x_i);
     }
     else{
          shared_mem[index] = 0.0;
     }
     }

     __syncthreads();

     // Convolution Code
     if ((ty < TILE_WIDTH) && (tx < TILE_WIDTH)){ // if dimensions
are within input bounds and M
     for (int m = 0; m < M; m++){    // Loop through output features
          float conv_val = 0.0; // Reset val for new output element
          for (int c = 0; c < C; c++){    // Loop through input
features
               // Loop through 2D convolution kernel/filter
               for (int ky = 0; ky < K; ky++){
                    for (int kx = 0; kx < K; kx++){
                    int index = c*SM_DIM*SM_DIM + (ty+ky)*SM_DIM +
(tx+kx);
                    conv_val += k4d(m, c, ky, kx) *
shared_mem[index];   // do convolution
                    }
               }
          }
```

4.4 Weight Matrix in Constant Memory

Similar to shared memory, CUDA provides another interesting feature to circumvent the bandwidth restriction on access to global memory. Unlike shared memory, constant memory is not on-chip, but is cached and designed to be accessed faster than global memory, in certain situations.

Constant memory slowed down our kernel. We suspect that the overhead of copying the weights from global to constant memory was greater than the speed boost we would've gotten from reading from global memory. The weights in global memory would also stored in a relatively fast L1 cache, thus resulting in no performance gains. It is worth noting, however, that the first layer (M=12, C = 1) had a slight speedup, but the second one did not (M=24, C=12). This is probably because the smaller layer had less cudaMemcpyToSymbol overhead.

We can clearly see from the NVVP timeline that the constant memory kernel is considerably slower than our other optimized kernels.

4.4.1 Timing Output

Op Time: 0.059142
Op Time: 0.872424
Correctness: 0.7653 Model: ece408

4.4.2 NVPROF Output

==353== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	93.22%	931.42ms	2	465.71ms	59.049ms	872.37ms	mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
	3.30%	32.971ms	20	1.6485ms	1.0240us	30.755ms	[CUDA memcpy HtoD]
	1.47%	14.706ms	2	7.3532ms	2.9287ms	11.778ms	void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>, mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul, mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
	0.79%	7.8598ms	1	7.8598ms	7.8598ms	7.8598ms	volta_sgemm_128x128_tn
	0.72%	7.2170ms	2	3.6085ms	25.376us	7.1917ms	void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)

0.44% 4.3831ms 1 4.3831ms 4.3831ms 4.3831ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

API calls: 39.00% 3.23393s 22 147.00ms 10.292us 1.65115s cudaStreamCreateWithFlags
27.98% 2.32000s 22 105.45ms 72.683us 2.31533s cudaMemGetInfo
19.42% 1.61049s 18 89.472ms 1.2540us 431.41ms cudaFree
11.41% 946.15ms 6 157.69ms 4.3470us 872.37ms cudaDeviceSynchronize
0.81% 66.956ms 9 7.4395ms 27.056us 30.968ms cudaMemcpy2DAsync
0.48% 39.627ms 66 600.41us 6.5380us 16.713ms cudaMalloc
0.45% 37.312ms 912 40.912us 439ns 19.195ms cudaFuncSetAttribute
0.23% 19.362ms 29 667.66us 3.2070us 10.927ms cudaStreamSynchronize

4.4.3 NVVP Timeline

4.4.4 Relevant Code

```
__constant__ float mask[24*12*5*5];
…
conv_val += mask[tz*C*K*K+c*K*K+ky*K+kx] * x4d(b, c, ty+ky, tx+kx);
```

Host code:

```
cudaMemcpyToSymbol(mask, k.dptr_, M*C*K*K*sizeof(float), 0,
cudaMemcpyHostToDevice);
```

4.5 Multiple Kernel Implementations for Different Layer Sizes (Multikernel)

We have two kernel functions, one for each layer size. The “small” kernel is for the M=12, C=1 layer, and the “large” kernel is for the M=24, C=12 layer. The two kernels have appropriate sizes for shared memory, and we see a performance boost for this. By allocating the right amount of resources, we waste less time writing unnecessary 0’s to the shared mem or checking bounds and reduce our op time.

4.5.1 Timing Output

Op Time: 0.030934
Op Time: 0.105163
Correctness: 0.7653 Model: ece408

4.5.2 NVPROF Output

==352== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 51.70% 105.13ms 1 105.13ms 105.13ms 105.13ms mxnet::op::forward_kernel_large(float*, float const *, float const *, int, int, int, int, int, int)
15.90% 32.331ms 20 1.6166ms 1.0560us 30.302ms [CUDA memcpy HtoD]
15.19% 30.883ms 1 30.883ms 30.883ms 30.883ms mxnet::op::forward_kernel_small(float*, float const *, float const *, int, int, int, int, int, int)
7.32% 14.891ms 2 7.4455ms 2.9188ms 11.972ms void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>, mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul, mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
3.91% 7.9609ms 1 7.9609ms 7.9609ms 7.9609ms volta_sgemm_128x128_tn
3.54% 7.2062ms 2 3.6031ms 24.576us 7.1816ms void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
2.14% 4.3506ms 1 4.3506ms 4.3506ms 4.3506ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,

cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

API calls:

43.27% 3.12584s 22 142.08ms 14.882us 1.59030s cudaStreamCreateWithFlags
31.03% 2.24181s 22 101.90ms 67.657us 2.23739s cudaMemGetInfo
21.27% 1.53674s 18 85.375ms 1.2290us 413.20ms cudaFree
2.09% 150.93ms 6 25.155ms 3.8290us 105.14ms cudaDeviceSynchronize
0.91% 65.532ms 9 7.2813ms 28.552us 30.455ms cudaMemcpy2DAsync
0.68% 48.780ms 912 53.487us 431ns 39.469ms cudaFuncSetAttribute
0.27% 19.572ms 29 674.90us 3.0820us 10.936ms cudaStreamSynchronize
0.22% 15.810ms 66 239.54us 6.4660us 3.0577ms cudaMalloc

4.5.3 NVVP Timeline

4.5.4 Relevant Code

```
#define TILE_WIDTH 16
#define MASK_WIDTH 5
#define MAX_M 24
#define MIN_M 12
#define MAX_C 12
#define MIN_C 1
#define SM_DIM (TILE_WIDTH+MASK_WIDTH-1) // shared memory dim
```

…

```
__global__ void forward_kernel_large(float *y, const float *x, const
float *k, const int B, const int M, const int C, const int H, const
int W, const int K)
{
     __shared__ float shared_mem[MAX_C*SM_DIM*SM_DIM];
…
__global__ void forward_kernel_small(float *y, const float *x, const
float *k, const int B, const int M, const int C, const int H, const
int W, const int K)
{
     __shared__ float shared_mem[MIN_C*SM_DIM*SM_DIM];
```

Host code:

```
// Call the kernel
     if (C==1){
     forward_kernel_small<<<gridDim, blockDim,
0>>>(y.dptr_,x.dptr_,k.dptr_, B,M,C,H,W,K);
     }
     else{
     forward_kernel_large<<<gridDim, blockDim,
0>>>(y.dptr_,x.dptr_,k.dptr_, B,M,C,H,W,K);
     }
```

4.6 Putting 3 Optimizations Together

We combined all of the optimizations that we have made thus far into one kernel and ran it to see our overall improvement.

4.6.1 Timing Output

Op Time: 0.027426
Op Time: 0.233445
Correctness: 0.7653 Model: ece408

4.6.2 NVPROF Output

==352== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	69.78%	233.38ms	1	233.38ms	233.38ms	233.38ms	mxnet::op::forward_kernel_large(float*, float const *, float const *, int, int, int, int, int, int)
	11.61%	38.835ms	20	1.9417ms	1.0880us	36.254ms	[CUDA memcpy HtoD]

```
                8.17%  27.324ms         1  27.324ms  27.324ms  27.324ms
mxnet::op::forward_kernel_small(float*, float const *, float const *, int, int, int, int, int, int)
                4.45%  14.878ms         2  7.4390ms  2.9155ms  11.962ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                2.35%  7.8720ms         1  7.8720ms  7.8720ms  7.8720ms
volta_sgemm_128x128_tn
                2.15%  7.2061ms         2  3.6031ms  24.832us  7.1813ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
                1.30%  4.3555ms         1  4.3555ms  4.3555ms  4.3555ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                0.12%  400.83us         1  400.83us  400.83us  400.83us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2, int)


API calls:   42.09%  3.40399s        22  154.73ms  15.150us  1.76411s
cudaStreamCreateWithFlags
                30.02%  2.42783s        22  110.36ms  71.383us  2.42554s  cudaMemGetInfo
                21.39%  1.73013s        18  96.118ms  1.2840us  473.91ms  cudaFree
                 3.41%  275.61ms         6  45.934ms  3.1460us  233.39ms  cudaDeviceSynchronize
                 0.96%  78.031ms         9  8.6701ms  30.357us  36.332ms  cudaMemcpy2DAsync
                 0.80%  64.431ms        66  976.22us  5.8590us  46.459ms  cudaMalloc
                 0.57%  46.384ms       912  50.859us     452ns  24.523ms  cudaFuncSetAttribute
                 0.35%  28.082ms       216  130.01us  1.3010us  26.555ms
cudaEventCreateWithFlags
                 0.24%  19.713ms        29  679.76us  3.6400us  10.895ms  cudaStreamSynchronize
```

4.6.3 NVVP Timeline

MILESTONE 5 (FINAL) REPORT

5.1 Introduction

For the milestone, we chose to implement 3 more of the optimizations in the suggested list on the project GitHub page. We will explain these optimizations and show the effect of said optimizations on the runtime of the neural network.

Once again, it is important to note that all our optimizations output the correct accuracy as mentioned on the project GitHub page, so this means that there is no change to the output of each convolution operation.

As recommended, we implemented these optimizations independently to better showcase the effect that each one has on the runtime of the convolution. Instead of just comparing our improvements to the baseline kernel mentioned in Section 4.2 of this report, we are adding optimizations on top of each other and tweaking parameters to find the true improvement. This is because parameters in a certain optimization might be different when applied to the base kernel instead of a kernel with other optimizations to yield the greatest speedup. We chose to implement the following optimizations:

- Kernel fusion for unrolling and matrix-multiplication
- Exploiting parallelism in input images and output channels
- Sweeping various parameters to find best values
- Tuning with restrict and loop unrolling

5.2 Kernel Fusion for Unrolling and Matrix-Multiplication

To increase performance, instead of having a separate kernel to unroll the filter and input matrices, we perform unrolling when loading the tile into shared memory in the matrix multiplication kernel. This is done by calculating the data indices such that the matrices are unrolled as they are loaded into shared memory. Thus, the work previously done by 2 kernels is fused into a single kernel. This improves performance because we eliminate the overhead needed to launch an extra kernel, as well as allocating and writing to extra memory to facilitate transferring the unrolled matrices between the two kernels. As shown in the NVVP timeline, we just have the overhead for launching 1 kernel instead of 2 kernels and extra memory operations.

5.2.1 Timing Output

Op Time: 0.045572
Op Time: 0.064764
Correctness: 0.7653 Model: ece408

5.2.2 NVPROF Output

```
==265== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.76%  110.22ms         2  55.110ms  45.501ms  64.720ms
mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                  18.76%  34.030ms        20  1.7015ms  1.0880us  31.852ms  [CUDA memcpy HtoD]
                   9.33%  16.924ms         2  8.4618ms  3.0649ms  13.859ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                   4.37%  7.9183ms         1  7.9183ms  7.9183ms  7.9183ms
volta_sgemm_128x128_tn
                   4.00%  7.2644ms         2  3.6322ms  24.767us  7.2396ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
                   2.43%  4.4041ms         1  4.4041ms  4.4041ms  4.4041ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                   0.24%  430.94us         1  430.94us  430.94us  430.94us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2, int)
                   0.04%  68.415us         1  68.415us  68.415us  68.415us  void
mshadow::cuda::SoftmaxKernel<int=8, float,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>>(mshadow::gpu,
int=2, unsigned int)
                   0.04%  64.863us        13  4.9890us  1.1840us  24.447us  void
mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2)
```

```
API calls:   42.04%  3.34171s       22  151.90ms  14.106us  1.71621s  cudaStreamCreateWithFlags
             32.90%  2.61493s       22  118.86ms  72.908us  2.60953s  cudaMemGetInfo
20.84%  1.65646s       18  92.026ms  1.2460us  442.58ms  cudaFree
              1.60%  127.22ms        6  21.203ms  12.163us  64.734ms  cudaDeviceSynchronize
              0.86%  68.219ms        9  7.5798ms  24.591us  31.856ms  cudaMemcpy2DAsync
              0.82%  65.263ms      216  302.14us  1.2870us  42.583ms  cudaEventCreateWithFlags
              0.30%  23.722ms      912  26.010us     440ns  10.917ms  cudaFuncSetAttribute
              0.25%  19.869ms       29  685.13us  5.6940us  11.026ms  cudaStreamSynchronize
              0.14%  11.345ms       66  171.90us  6.4640us  1.7428ms  cudaMalloc
              0.09%  7.4554ms        4  1.8639ms  1.2294ms  2.4802ms  cudaGetDeviceProperties
              0.05%  3.6243ms       12  302.02us  7.1360us  3.0850ms  cudaMemcpy
              0.03%  2.7242ms      375  7.2640us     413ns  382.66us  cuDeviceGetAttribute
              0.02%  1.2958ms        9  143.98us  9.4150us  1.0333ms  cudaMemsetAsync
              0.01%  1.0011ms        2  500.56us  54.646us  946.48us  cudaHostAlloc
              0.01%  741.00us       27  27.444us  10.200us  68.685us  cudaLaunchKernel
              0.01%  626.13us        4  156.53us  97.395us  276.55us  cuDeviceTotalMem
              0.01%  566.84us        4  141.71us  75.494us  248.02us  cudaStreamCreate
              0.01%  435.44us      202  2.1550us     823ns  44.130us  cudaDeviceGetAttribute
```

5.2.3 NVVP Timeline

5.2.4 Relevant Code

Kernel code:

```
...
int numMatACols = C*K*K; // = numMatBRows
...
// Convolution Code
int num_iter = ceil(numMatACols/(1.0*TILE_WIDTH));
```

```
for (int i = 0; i < num_iter; i++)
{
    int temp_col = i * TILE_WIDTH + tx;
    int temp_row = i * TILE_WIDTH + ty;

    // filter tensor
    int k_m = row;
    int k_c = temp_col / (K*K);
    int k_h = (temp_col % (K*K)) / K;
    int k_w = (temp_col % (K*K)) % K;

    // input tensor
    int x_b = b;
    int x_c = temp_row / (K*K);
    int x_p = (temp_row % (K*K)) / K;
    int x_q = (temp_row % (K*K)) % K;
    int x_h = col / W_out;
    int x_w = col % W_out;

    if (temp_col < numMatACols && row < M)
        tileMatA[ty][tx] = k4d(k_m, k_c, k_h, k_w);
    else
        tileMatA[ty][tx] = 0.0;

    if (temp_row < numMatACols && col < H_out * W_out)
        tileMatB[ty][tx] = x4d(x_b, x_c, x_h + x_p, x_w + x_q);
    else
        tileMatB[ty][tx] = 0.0;

    __syncthreads();

    for (int q = 0; q < TILE_WIDTH; q++)
        conv_val += tileMatA[ty][q] * tileMatB[q][tx];

    __syncthreads();
}

// output tensor
int y_b = b;
int y_m = row;
int y_h = col / W_out;
int y_w = col % W_out;

if (row < M && col < W_out * H_out)
```

```
        y4d(y_b, y_m, y_h, y_w) = conv_val;
```

Host code:

```
...
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(ceil((H_OUT*W_OUT)/(1.0*TILE_WIDTH)),
ceil((M)/(1.0*TILE_WIDTH)), B);
// call kernel
...
```

5.3 - Exploiting Parallelism in Input Images and Output Channels

We exploited parallelism in our optimized kernel by threading on M (number of output channels), and B (number of input images). As seen in our kernel launch code, for both sizes of kernels, the Y dimension of the grid depends on a tiled parameter of M, and the Z dimension of the grid is B.

We wanted to exploit parallelism on input images since: (1) there are 10,000 images, so looping through B in the kernel would be extremely slow (what we did in milestone 3. Section 3.1 shows that the total Op Time was around 370 ms). (2) each image is the same size, so the same kind of work must be done on it, making this an ideal parameter to exploit parallely. (3) The dimension of the input is B * C * H * W. Thus, we get no memory coalescence by having the same thread work on multiple images.

We wanted to exploit parallelism on output channels since the same work needs to be done on every output channel. We leveraged this by setting out tile width according to the M values which were known to be 12 in the first layer (for the small kernel) and 24 in the second layer (for the large kernel). This arrangement is very intuitive and worked really well. Each thread writes to the output and no threads need to be turned off. Thus it is resource efficient.

We had actually noticed this potential improvement after completing Milestone 3, so we implemented using the threads in the Z dimension to process different images in parallel, and using threads to process the image channels in parallel instead of each thread processing channels through for loops. This became our new baseline kernel as discussed in Section 4.2. In this section, we optimized this further by taking advantage of tiling.

5.3.1 Timing Output

Op Time: 0.026548
Op Time: 0.046994
Correctness: 0.7653 Model: ece408

5.3.2 NVPROF Output

==266== Profiling result:

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   32.14%  46.975ms         1  46.975ms  46.975ms  46.975ms
mxnet::op::forward_kernel_large(float*, float const *, float const *, int, int, int, int, int, int)
                  24.27%  35.469ms        20  1.7734ms  1.1200us  33.154ms  [CUDA memcpy HtoD]
                  18.13%  26.499ms         1  26.499ms  26.499ms  26.499ms
mxnet::op::forward_kernel_small(float*, float const *, float const *, int, int, int, int, int, int)
                  11.57%  16.909ms         2  8.4544ms  3.0651ms  13.844ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                   5.42%  7.9183ms         1  7.9183ms  7.9183ms  7.9183ms
volta_sgemm_128x128_tn
                   5.02%  7.3307ms         2  3.6654ms  25.696us  7.3050ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
                   3.01%  4.3994ms         1  4.3994ms  4.3994ms  4.3994ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                   0.31%  452.48us         1  452.48us  452.48us  452.48us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2, int)


API calls:
41.71%  3.23193s        22  146.91ms  14.966us  1.65991s  cudaStreamCreateWithFlags
                  32.69%  2.53254s        22  115.12ms  68.282us  2.52803s  cudaMemGetInfo
                  21.74%  1.68419s        18  93.566ms  1.2200us  441.91ms  cudaFree
                   1.17%  90.400ms         6  15.067ms  3.0110us  46.979ms  cudaDeviceSynchronize
                   0.92%  71.309ms         9  7.9232ms  38.307us  33.192ms  cudaMemcpy2DAsync
                   0.80%  62.119ms       912  68.112us     439ns  11.141ms  cudaFuncSetAttribute
                   0.40%  30.729ms        66  465.59us  6.6360us  17.906ms  cudaMalloc
                   0.26%  20.016ms        29  690.19us  2.4070us  11.102ms  cudaStreamSynchronize
                   0.12%  9.3476ms        12  778.96us  6.7800us  8.9002ms  cudaMemcpy
                   0.09%  6.8663ms         4  1.7166ms  426.36us  2.2094ms
cudaGetDeviceProperties
```

```
                0.03%  2.6189ms       375  6.9830us     411ns  333.61us  cuDeviceGetAttribute
                0.02%  1.8248ms       216  8.4470us  1.2690us  466.65us
cudaEventCreateWithFlags
```

5.3.3 NVVP Timeline

5.3.4 Relevant Code

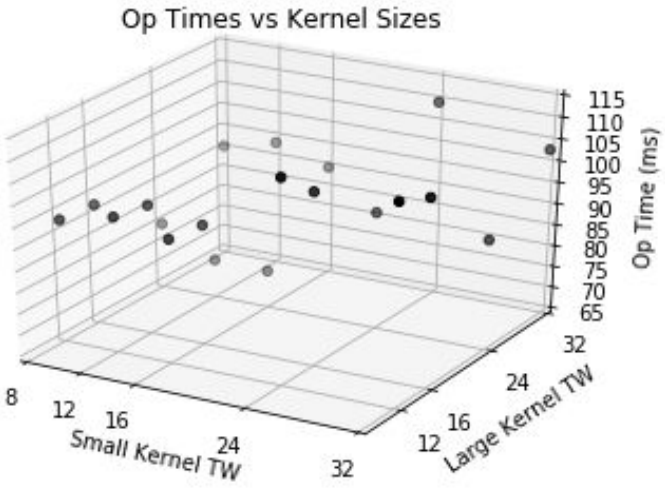
```
#define TILE_WIDTH_LARGE 24
#define TILE_WIDTH_SMALL 12
```

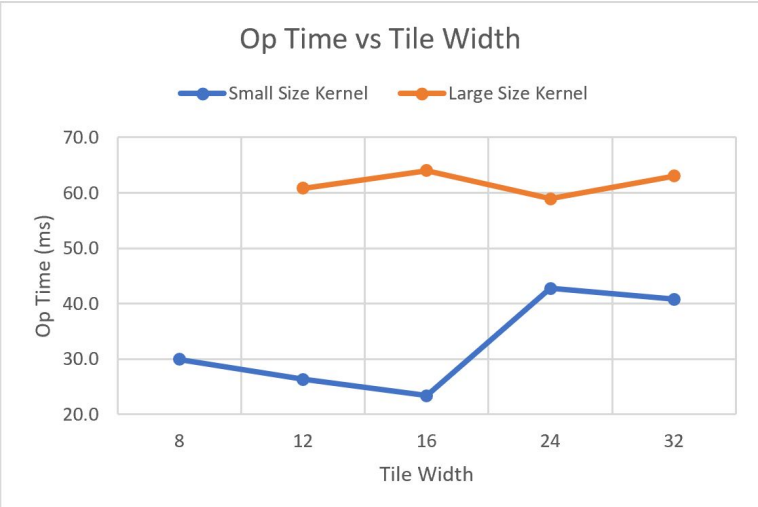
Host code:

```
// Call the kernel
if (C==1){
        dim3 blockDim(TILE_WIDTH_SMALL, TILE_WIDTH_SMALL, 1);
        dim3 gridDim(ceil((H_OUT*W_OUT)/(1.0*TILE_WIDTH_SMALL)),
ceil((M)/(1.0*TILE_WIDTH_SMALL)), B);

        // Call the kernel
        forward_kernel_small<<<gridDim, blockDim,
0>>>(y.dptr_,x.dptr_,k.dptr_, B,M,C,H,W,K);
    }
    else{
        dim3 blockDim(TILE_WIDTH_LARGE, TILE_WIDTH_LARGE, 1);
        dim3 gridDim(ceil((H_OUT*W_OUT)/(1.0*TILE_WIDTH_LARGE)),
ceil((M)/(1.0*TILE_WIDTH_LARGE)), B);

        // Call the kernel
        forward_kernel_large<<<gridDim, blockDim,
0>>>(y.dptr_,x.dptr_,k.dptr_, B,M,C,H,W,K);
    }
```

5.4 Sweeping Parameters to Find Best Values

To improve our runtime, we swept through parameters for the small and large tile widths for the corresponding kernel, as implemented from our multi-kernel optimization. We tried tile widths of sizes 8, 12, 16, 24, and 32 for the small kernel and sizes 12, 16, 24, 32 for the large kernel. We chose to test tile widths that were powers of 2 and a couple of values in between to maximize the usage of the tiles. As shown in the graphs below, the overall op time was lowest when using a small tile width of 16 and a large tile width of 24. We saw in section 5.3 how small tile width of 12 and large tile width of 24 would be a good choice, but found that having the small tile width as 16 improved our performance. This makes sense because the boost from increased reuse of the input matrix elements (by having more threads to load the input matrix and larger shared memory) outweighs the cost of having 4 threads that do not write to the output channels. These parameters gave us our best Op Time.





Kernel		Op Times (s)		
Small Size	Large Size	Op Small	Op Large	Total
8	12	0.029922	0.062788	0.092710
8	16	0.029910	0.062204	0.092114
8	24	0.030098	0.049235	0.079333
8	32	0.029897	0.060317	0.090214
12	12	0.026274	0.069462	0.095736
12	16	0.026441	0.067913	0.094354
12	24	0.026124	0.046834	0.072958
12	32	0.026427	0.066855	0.093282
16	12	0.023350	0.069720	0.093070
16	16	0.023492	0.068635	0.092127
16	24	0.023500	0.049244	0.072744
16	32	0.023353	0.066219	0.089572
24	12	0.042873	0.068766	0.111639
24	16	0.042870	0.061578	0.104448
24	24	0.042641	0.048913	0.091554
24	32	0.042608	0.066660	0.109268
32	12	0.040896	0.070158	0.111054
32	16	0.040681	0.067166	0.107847
32	24	0.040911	0.049059	0.089970
32	32	0.040678	0.062044	0.102722

Best Params				
16	24	0.0235	0.049244	0.072744
Ranking Op Time (ms)				70.684

5.4.1 Timing Output

Op Time: 0.023880
Op Time: 0.047964
Correctness: 0.7653 Model: ece408

5.4.2 NVPROF Output

==264== Profiling result:

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   34.42%  47.944ms         1  47.944ms  47.944ms  47.944ms
mxnet::op::forward_kernel_large(float*, float const *, float const *, int, int, int, int, int, int)
                  23.00%  32.029ms        20  1.6015ms  1.0880us  29.884ms  [CUDA memcpy HtoD]
                  17.12%  23.848ms         1  23.848ms  23.848ms  23.848ms
mxnet::op::forward_kernel_small(float*, float const *, float const *, int, int, int, int, int, int)
                  10.61%  14.777ms         2  7.3886ms  2.9253ms  11.852ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                   6.10%  8.4950ms         1  8.4950ms  8.4950ms  8.4950ms
volta_sgemm_128x128_tn
                   5.17%  7.2056ms         2  3.6028ms  25.376us  7.1802ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
                   3.13%  4.3573ms         1  4.3573ms  4.3573ms  4.3573ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                   0.29%  409.50us         1  409.50us  409.50us  409.50us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2, int)


API calls:
41.63%  3.05899s        22  139.05ms  15.055us  1.56945s  cudaStreamCreateWithFlags
                  33.60%  2.46880s        22  112.22ms  70.062us  2.46418s  cudaMemGetInfo
```

```
 20.99%  1.54239s       18  85.688ms  1.2250us  408.67ms  cudaFree
  1.18%  86.585ms        6  14.431ms  2.8350us  47.948ms  cudaDeviceSynchronize
  0.88%  64.902ms        9  7.2113ms  28.238us  30.037ms  cudaMemcpy2DAsync
  0.74%  54.707ms      912  59.985us     431ns  26.032ms  cudaFuncSetAttribute
  0.29%  21.393ms       66  324.13us  6.6920us  4.8030ms  cudaMalloc
  0.27%  20.134ms       29  694.28us  2.4010us  10.955ms  cudaStreamSynchronize
  0.25%  18.046ms      216  83.548us  1.2650us  8.0995ms
cudaEventCreateWithFlags
  0.06%  4.2093ms        4  1.0523ms  339.42us  1.5823ms
cudaGetDeviceProperties
  0.03%  2.1682ms      375  5.7810us     410ns  273.44us  cuDeviceGetAttribute
```

5.4.3 NVVP Timeline

5.4.4 Relevant Code

5.4.4.1 Relevant Code from Project File

```
#define TILE_WIDTH_LARGE 24
#define TILE_WIDTH_SMALL 16
```

5.4.4.2 Python Code to plot the graphs from the excel table of values (as shown in section 5.4)

```
# imports
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

# data array copied over from the excel sheet
data = np.array([
[[ 8],[12],[0.09271 ]],[[ 8],[16],[0.092114]],[[ 8],[24],[0.079333]],
[[ 8],[32],[0.090214]],[[12],[12],[0.095736]],[[12],[16],[0.094354]],
[[12],[24],[0.072958]],[[12],[32],[0.093282]],[[16],[12],[0.09307 ]],
[[16],[16],[0.092127]],[[16],[24],[0.072744]],[[16],[32],[0.089572]],
```

```
[[24],[12],[0.111639]],[[24],[16],[0.104448]],[[24],[24],[0.091554]],
[[24],[32],[0.109268]],[[32],[12],[0.111054]],[[32],[16],[0.107847]],
[[32],[24],[0.08997 ]],[[32],[32],[0.102722]]])

X = data[:,0]         # tile widths for small kernel
Y = data[:,1]         # tile widths for large kernel
Z = data[:,2]*1000    # total op times in ms

# plot graph
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
ax.scatter3D(X,Y,Z, c='black', marker='o')
# label graph and save it
ax.set_xlabel('Small Kernel TW')
ax.set_ylabel('Large Kernel TW')
ax.set_zlabel('Op Time (ms)')
ax.set_xlim3d([8,32])
ax.set_ylim3d([8, 32])
ax.set_zlim3d([65,115])
ax.set_xticks([8,12,16,24,32])
ax.set_yticks([12,16,24,32])
ax.set_zticks([65,70,75,80,85,90,95,100,105,110,115])
ax.set_title('Op Times vs Kernel Sizes')
plt.savefig('graph_3d.png')
```

5.5 Tuning with Restrict and Loop Unrolling (with Multi Kernel)

We used this in the convolutional kernel to great effect. By unrolling the inner-most loop, we were able to get a better runtime (it performed better than multi-kernel in section 4.5). The inner loop was for the X dim of the convolution filter. The size is always K=5, so we copied the multiplication and addition to conv_val statement 5 times. We found that manually copying the code performed better than #pragma unroll by a few milliseconds (timing not provided as it was only slightly faster). We think that this is because the programmer explicitly stating what they want out of the unrolling is better than the compiler guessing what the programmer wants in this case. It was also a small loop. We tried adding #pragma unroll to other loops individually and concurrently, but the initially described configuration performed the best. This optimization was not very helpful for the fusion matrix multiply kernel. This is probably because shared memory matrix multiply is already a fairly efficient algorithm and doesn't get much gain from loop unrolling since there aren't many loops (direct matmul vs looping to multiply).

5.5.1 Timing Output

New Inference
Op Time: 0.014937
Op Time: 0.070421

5.5.2 NVPROF Output

```
==264== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:   45.51%  70.393ms         1  70.393ms  70.393ms  70.393ms
mxnet::op::forward_kernel_large(float*, float const *, float const *, int, int, int, int, int, int)
                  21.79%  33.702ms        20  1.6851ms  1.0880us  31.490ms  [CUDA memcpy HtoD]
                   9.62%  14.879ms         1  14.879ms  14.879ms  14.879ms
mxnet::op::forward_kernel_small(float*, float const *, float const *, int, int, int, int, int, int)
                   9.58%  14.822ms         2  7.4110ms  2.9220ms  11.900ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                   5.63%  8.7039ms         1  8.7039ms  8.7039ms  8.7039ms
volta_sgemm_128x128_tn
                   4.66%  7.2078ms         2  3.6039ms  25.823us  7.1820ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float,
dimArray, reducedDivisorArray)
                   2.82%  4.3576ms         1  4.3576ms  4.3576ms  4.3576ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                   0.26%  406.59us         1  406.59us  406.59us  406.59us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>,
mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int,
mshadow::Shape<int=2>, int=2, int)


API calls:   41.20%  3.27691s        22  148.95ms  15.324us  1.70117s
cudaStreamCreateWithFlags
                  33.69%  2.67948s        22  121.79ms  71.907us  2.67536s  cudaMemGetInfo
                  20.80%  1.65423s        18  91.902ms  1.2490us  442.30ms  cudaFree
                   1.35%  107.36ms       216  497.06us  1.3130us  105.32ms
cudaEventCreateWithFlags
                   1.26%  100.13ms         6  16.688ms  4.2680us  70.400ms  cudaDeviceSynchronize
                   0.86%  68.378ms         9  7.5976ms  30.821us  31.717ms  cudaMemcpy2DAsync
```

```
                0.25%  20.082ms       29  692.50us  3.4460us  10.709ms  cudaStreamSynchronize
                0.23%  18.202ms      912  19.958us     440ns  9.2931ms  cudaFuncSetAttribute
                0.19%  15.319ms       66  232.11us  6.6730us  4.8858ms  cudaMalloc
                0.07%  5.2087ms        4  1.3022ms  434.08us  1.9509ms
cudaGetDeviceProperties
                0.03%  2.6270ms      375  7.0050us     413ns  339.11us  cuDeviceGetAttribute
```

5.5.3 NVVP Timeline

5.5.4 Relevant Code

Kernel code:

```
__global__ void forward_kernel_small(float* __restrict__ y, const
float* __restrict__ x, const float* __restrict__ k, const int B,
const int M, const int C, const int H, const int W, const int K)
{
...
for (int ky = 0; ky < K; ky++){
                    conv_val += k4d(m, c, ky, 0) * shared_mem[index];
// do convolution
                    conv_val += k4d(m, c, ky, 1) *
shared_mem[index+1];   // do convolution
                    conv_val += k4d(m, c, ky, 2) *
shared_mem[index+2];   // do convolution
                    conv_val += k4d(m, c, ky, 3) *
shared_mem[index+3];   // do convolution
                    conv_val += k4d(m, c, ky, 4) *
shared_mem[index+4];   // do convolution
                    index += SM_DIM_SMALL;
 }
...
}
```

5.6 Final Remarks

5.6.1 Overview and Kernel Description

Overall, by implementing various optimizations, we were able to greatly increase the performance of our image convolution kernel. Our best kernel was the one which combined Kernel Fusion for Unrolling and Matrix-Multiplication (section 5.2) with shared memory (incorporated in section 5.2 and also used in section 4.3), multiple kernels for different layer sizes (section 4.5), and incorporating values found from sweeping (section 5.4). Section 5.4 builds off of the “exploiting parallelism” optimization described in section 5.3. This kernel is also described above in some detail in section 5.4. We called this the “Multi-fusion” kernel.

5.6.2 Results

Of the 7 optimizations we implemented, using constant memory and restrict plus loop unrolling slowed down the total op time of the multi-fusion kernel, so we left them out of our final version used to compete for speed in the rankings. Our unoptimized kernel from Milestone 3 had op times of 0.051133 + 0.326978 = 0.378111 seconds. Our most optimal kernel, Multi-fusion (described above), had op times of 0.023880 + 0.047964 = 0.071844 seconds, while maintaining full correctness. That is approximately a 5.263x speedup.

5.6.3 Identifying Optimization Opportunity

We mainly referred to the list of possible optimizations given on the project GitHub page. We related them to course material and labs from the semester and set about implementing them. All relevant details are provided in the writeup provided for each optimization. We discussed our approach to them, the results, and justifications.

5.6.4 References

We used course material (lecture slides, past exams, textbook) for this project. If we used any external sources, they have been mentioned.

5.6.5 Teamwork

The team functioned smoothly. The initial checkpoints were very straight-forward so the team met up and collaborated to finish them (code, report, nvprof, nvvp). In the later checkpoints, the team members tried different optimizations and we picked whichever ones were successful. We organized the team into code and report. Advaith focussed more on the code, and Adit and Murugan focussed more on the report and nvvp, as this had more components. The team would be flexible and help write/debug code or finish/submit the report as necessary. This was true for all of the optimizations.

5.6.6 Directory Structure

We have included all of our optimizations in the submission file under the ece408_src folder. The optimizations can be found in the following files:

1) Shared Memory convolution: shared_mem.cuh, new-forward.cuh, multi-kernel.cuh, restrict_loop_unroll.cuh
2) Kernel fusion for unrolling and matrix-multiplication: kernel_fusion.cuh, new-forward.cuh
3) Weight matrix (kernel values) in constant memory: constant_mem.cuh
4) Tuning with restrict and loop unrolling: restrict_loop_unroll.cuh
5) Sweeping various parameters to find best values: new-forward.cuh
6) Exploiting parallelism in input images, input channels, and output channels: new-forward.cuh, multi_kernel.cuh
7) Multiple kernel implementations for different layer sizes: new-forward.cuh, multi_kernel.cuh

Our baseline kernel is included as base.cuh