ECE 408 PROJECT REPORT

University of Illinois at Urbana-Champaign

FALL 2019

Submitted by

TEAM: OPTIMIZERS

NAME: Kevin White, NETID: kevinjw2,

UIN: 654368454

NAME: Anjana S. Kumar, NETID: anjanas3,

UIN: 652193822

NAME: Nishqa Sharma, NETID: nsharm13,

UIN: 671462226

Contents

List of Figures

1 Milestone 2

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

- volta_scudnn_128x64_relu_interior_nn_v1
- volta_gcgemm_64x32_nt
- void fft2d_c2r_32x32
- volta_sgemm_128x128_tn
- void op_generic_tensor_kernel
- void fft2d_r2c_32x32

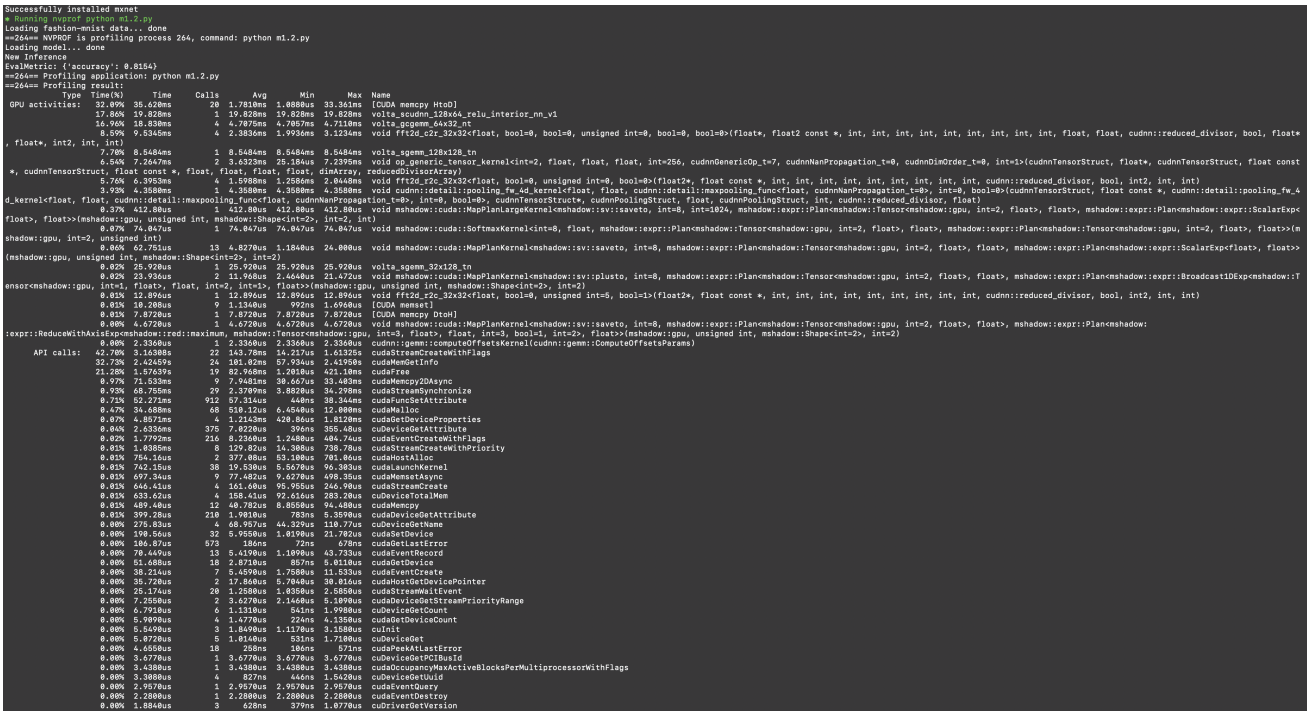

Figure 1: NV Profile analysis

1.2 List of all CUDA API calls that collectively consume more than 90% of the program time

- cudaStreamCreateWithFlags
- cudaMemGetInfo
- cudaFree

1.3 Difference between Kernels and API calls

The CUDA kernel is a C function run by each thread and launched on the GPU in parallel. The number of threads is decided by the programmer in the "execution configuration syntax" in the kernel invocation code written in the host code, under constraints offered by the specific hardware in use.

In contrast, the CUDA API contains extensions to the C programming language that assist us in easily writing kernel code that can run on the device. It consists of C language extensions, such as function execution space specifiers, variable memory specifiers, built in vector types and variables, and several functions, such as memory fence functions, synchronization functions, math functions, surface functions, etc. The runtime library consists of C functions that execute on the host to allocate and de-allocate device memory, transfer data between host memory and device memory, and manage systems with multiple devices, among other functions. It also consists of a lower level C-API, which includes low level concepts such as CUDA context and CUDA modules, which most applications do not require that level of control.

1.4 Output of rai running MXNet on the CPU

```
Successfully installed mxnet
* Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
17.07user 4.86system 0:08.97elapsed 244%CPU (0avgtext+0avgdata 6046696maxresident)k
0inputs+2824outputs (0
major+1604330minor)pagefaults 0swaps
```

Figure 2: MXNet on CPU

1.4.1 Program run time

The total time elapsed- 8.97s

1.5 Output of rai running MXNet on the GPU

```
Successfully installed mxnet
* Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
5.00user 3.17system 0:04.68elapsed 174%CPU (0avgtext+0avgdata
2966748maxresident)k
0inputs+1720outputs (0major+731975minor)pagefaults 0swaps
```

Figure 3: MXNet on GPU

1.5.1 Program run time

The total time elapsed- 4.68s

1.6 CPU implementation

```
Successfully installed mxnet
* Running /usr/bin/time python m2.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 10.850050
Op Time: 59.249537
Correctness: 0.7653 Model: ece408
83.25user 8.36system 1:13.73elapsed 124%CPU (0avgtext+0avgdata 6044788maxresident)k
0inputs+2824outputs (0major+2312042minor)pagefaults 0swaps
```

Figure 4: MXNet for CPU Implementation

1.6.1 Program execution time

The total time elapsed- 1m 13.73s

1.6.2 Op Times

- Op Time for layer 1: 10.850050s

- Op Time for layer 2: 59.249537s

2 Milestone 3

2.1 Dataset 100

- Op Time for layer 1: 0.000309s
- Op Time for layer 2: 0.001064s
- Correctness: 0.69
- Total time elapsed: 4.47s

2.2 Dataset 1000

- Op Time for layer 1: 0.003692s
- Op Time for layer 2: 0.011841s
- Correctness: 0.697
- Total time elapsed: 4.53s

2.3 Dataset 10000

- Op Time for layer 1: 0.057152s
- Op Time for layer 2: 0.121658s
- Correctness: 0.7094
- Total time elapsed: 4.80s

2.4 Demo of NVPROF for dataset 100

```
Correctness: 0.69 Model: ece408
==353== Profiling application: python m3.1.py 100
==353== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.57%  2.6280ms        20  131.40us  1.1200us  2.2349ms  [CUDA memcpy HtoD]
                   29.56%  1.3262ms         2  663.11us  274.08us  1.0522ms  mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                    4.13%  185.09us         1  185.09us  185.09us  185.09us  volta_sgemm_32x32_sliced1x4_tn
                    2.33%  104.64us         2  52.319us  28.064us  76.575us  void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>,
hadow::gpu, int=4, float>, float, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4)
                    1.73%  77.440us         2  38.720us  2.1760us  75.264us  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)
                    1.54%  69.312us        14  4.9500us  1.1840us  24.384us  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)
                    1.18%  52.928us         1  52.928us  52.928us  52.928us  void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, b
ool=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*, cudnnPoolingS
truct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                    0.38%  16.895us         1  16.895us  16.895us  16.895us  volta_sgemm_128x32_tn
                    0.22%  9.8230us         9  1.0910us     992ns  1.3760us  [CUDA memset]
                    0.10%  4.3840us         1  4.3840us  4.3840us  4.3840us  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::ReduceWithAxisExp<mshadow::red::maximum, mshadow::Tensor<mshadow::gpu, int=3, float>, float, int=3, bool=1, int=2>, float>>(mshadow::gpu, unsigned int, mshadow::
Shape<int=2>, int=2)
                    0.09%  4.2240us         2  2.1120us  1.5040us  2.7200us  void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>,
float>, mshadow::expr::Plan<mshadow::expr::Broadcast1DExp<mshadow::Tensor<mshadow::gpu, int=1, float>, float, int=2, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
                    0.09%  4.0640us         1  4.0640us  4.0640us  4.0640us  void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshado
w::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>>(mshadow::gpu, int=2, unsigned int)
                    0.08%  3.5840us         1  3.5840us  3.5840us  3.5840us  [CUDA memcpy DtoH]
      API calls:   49.49%  3.87109s        22  175.96ms  14.830us  1.96336s  cudaStreamCreateWithFlags
                   28.98%  2.26675s        22  103.03ms  94.663us  2.26352s  cudaMemGetInfo
                   20.48%  1.60156s        18  88.976ms  1.2040us  423.59ms  cudaFree
                    0.37%  28.555ms         6  4.7591ms  1.7080us  26.768ms  cudaEventCreate
                    0.34%  26.879ms       912  29.472us     423ns  8.5736ms  cudaFuncSetAttribute
                    0.07%  5.5644ms         9  618.26us  13.530us  2.2728ms  cudaMemcpy2DA
sync
                    0.06%  4.6443ms        66  70.368us  6.2760us  403.07us  cudaMalloc
                    0.06%  4.5863ms         4  1.1466ms  746.32us  1.5442ms  cudaGetDeviceProperties
                    0.03%  2.6102ms       375  6.9600us     394ns  332.24us  cuDeviceGetAttribute
                    0.03%  2.3251ms       216  10.764us  1.2020us  910.40us  cudaEventCreateWithFlags
                    0.02%  1.4506ms         6  241.76us  6.7040us  1.0557ms  cudaDeviceSynchronize
                    0.01%  951.22us         2  475.61us  40.344us  910.87us  cudaHostAlloc
                    0.01%  781.41us         4  195.35us  93.849us  402.68us  cuDeviceTotalMem
                    0.01%  695.28us        29  23.975us  3.2440us  170.27us  cudaStreamSynchronize
                    0.01%  601.42us         8  75.177us  23.260us  349.62us  cudaStreamCreateWithPriority
                    0.01%  595.64us        27  22.060us  11.795us  59.528us  cudaLaunchKernel
                    0.01%  520.70us        12  43.391us  8.7090us  122.69us  cudaMemcpy
                    0.01%  500.37us         4  125.09us  68.738us  226.06us  cudaStreamCreate
                    0.00%  367.94us       202  1.8210us     791ns  5.1220us  cudaDeviceGetAttribute
                    0.00%  276.95us         4  69.238us  46.459us  103.48us  cuDeviceGetName
                    0.00%  269.36us         9  29.928us  13.297us  50.878us  cudaMemsetAsync
                    0.00%  145.85us        29  5.0290us  1.0370us  18.481us  cudaSetDevice
                    0.00%  101.66us       557     182ns      73ns     976ns  cudaGetLastError
                    0.00%  62.491us         2  31.245us  3.6630us  58.828us  cudaHostGetDevicePointer
                    0.00%  51.702us        18  2.8720us     851ns  5.5560us  cudaGetDevice
                    0.00%  16.394us         3  5.4640us  4.7920us  6.0530us  cudaEventRecord
                    0.00%  7.3740us         6  1.2290us     610ns  2.6620us  cuDeviceGetCount
                    0.00%  6.0940us         5  1.2180us     610ns  2.1820us  cuDeviceGet
                    0.00%  5.9410us         1  5.9410us  5.9410us  5.9410us  cudaEventQuery
                    0.00%  5.6040us         1  5.6040us  5.6040us  5.6040us  cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
                    0.00%  5.2550us         3  1.7510us  1.1240us  2.9400us  cuInit
                    0.00%  5.1370us        20     256ns     134ns     433ns  cudaPeekAtLastError
                    0.00%  4.9940us         2  2.4970us  2.1960us  2.7980us  cudaDeviceGetStreamPriorityRange
                    0.00%  4.7730us         1  4.7730us  4.7730us  4.7730us  cuDeviceGetPCIBusId
                    0.00%  3.5250us         4     881ns     463ns  1.5210us  cuDeviceGetUuid
                    0.00%  2.6540us         4     663ns     236ns  1.4140us  cudaGetDeviceCount
```

Figure 5: NVProf analysis part 1

```
                  0.09%  4.2240us         2  2.1120us  1.5040us  2.7200us  void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>,
float>, mshadow::expr::Plan<mshadow::expr::Broadcast1DExp<mshadow::Tensor<mshadow::gpu, int=1, float>, float, int=2, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
                  0.09%  4.0640us         1  4.0640us  4.0640us  4.0640us  void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshado
w::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>>(mshadow::gpu, int=2, unsigned int)
                  0.08%  3.5840us         1  3.5840us  3.5840us  3.5840us  [CUDA memcpy DtoH]
      API calls:   49.49%  3.87109s        22  175.96ms  14.830us  1.96336s  cudaStreamCreateWithFlags
                  28.98%  2.26675s        22  103.03ms  94.663us  2.26352s  cudaMemGetInfo
                  20.48%  1.60156s        18  88.976ms  1.2040us  423.59ms  cudaFree
                   0.37%  28.555ms         6  4.7591ms  1.7080us  26.768ms  cudaEventCreate
                   0.34%  26.879ms       912  29.472us     423ns  8.5736ms  cudaFuncSetAttribute
                   0.07%  5.5644ms         9  618.26us  13.530us  2.2728ms  cudaMemcpy2DA
sync
                   0.06%  4.6443ms        66  70.368us  6.2760us  403.07us  cudaMalloc
                   0.06%  4.5863ms         4  1.1466ms  746.32us  1.5442ms  cudaGetDeviceProperties
                   0.03%  2.6102ms       375  6.9600us     394ns  332.24us  cuDeviceGetAttribute
                   0.03%  2.3251ms       216  10.764us  1.2020us  910.40us  cudaEventCreateWithFlags
                   0.02%  1.4506ms         6  241.76us  6.7040us  1.0557ms  cudaDeviceSynchronize
                   0.01%  951.22us         2  475.61us  40.344us  910.87us  cudaHostAlloc
                   0.01%  781.41us         4  195.35us  93.849us  402.68us  cuDeviceTotalMem
                   0.01%  695.28us        29  23.975us  3.2440us  170.27us  cudaStreamSynchronize
                   0.01%  601.42us         8  75.177us  23.260us  349.62us  cudaStreamCreateWithPriority
                   0.01%  595.64us        27  22.060us  11.795us  59.528us  cudaLaunchKernel
                   0.01%  520.70us        12  43.391us  8.7090us  122.69us  cudaMemcpy
                   0.01%  500.37us         4  125.09us  68.738us  226.06us  cudaStreamCreate
                   0.00%  367.94us       202  1.8210us     791ns  5.1220us  cudaDeviceGetAttribute
                   0.00%  276.95us         4  69.238us  46.459us  103.48us  cuDeviceGetName
                   0.00%  269.36us         9  29.928us  13.297us  50.878us  cudaMemsetAsync
                   0.00%  145.85us        29  5.0290us  1.0370us  18.481us  cudaSetDevice
                   0.00%  101.66us       557     182ns      73ns     976ns  cudaGetLastError
                   0.00%  62.491us         2  31.245us  3.6630us  58.828us  cudaHostGetDevicePointer
                   0.00%  51.702us        18  2.8720us     851ns  5.5560us  cudaGetDevice
                   0.00%  16.394us         3  5.4640us  4.7920us  6.0530us  cudaEventRecord
                   0.00%  7.3740us         6  1.2290us     610ns  2.6620us  cuDeviceGetCount
                   0.00%  6.0940us         5  1.2180us     610ns  2.1820us  cuDeviceGet
                   0.00%  5.9410us         1  5.9410us  5.9410us  5.9410us  cudaEventQuery
                   0.00%  5.6040us         1  5.6040us  5.6040us  5.6040us  cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
                   0.00%  5.2550us         3  1.7510us  1.1240us  2.9400us  cuInit
                   0.00%  5.1370us        20     256ns     134ns     433ns  cudaPeekAtLastError
                   0.00%  4.9940us         2  2.4970us  2.1960us  2.7980us  cudaDeviceGetStreamPriorityRange
                   0.00%  4.7730us         1  4.7730us  4.7730us  4.7730us  cuDeviceGetPCIBusId
                   0.00%  3.5250us         4     881ns     463ns  1.5210us  cuDeviceGetUuid
                   0.00%  2.6540us         4     663ns     236ns  1.4140us  cudaGetDeviceCount
                   0.00%  1.9720us         3     657ns     331ns  1.2870us  cuDriverGetVersion
* Running nvprof -o timeline100.nvvp python m3.1.py 100
Loading fashion-mnist data... done
==449== NVPROF is profiling process 449, command: python m3.1.py 100
Loading model... done
```

Figure 6: NVProf analysis part 2

2.5 Demo of running NVProf

```
0inputs+45760outputs (0major+637955minor)pagefaults 0swaps
* Running nvprof -o timeline.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==352== NVPROF is profiling process 352, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000295
Op Time: 0.001080
Correctness: 0.69 Model: ece408
==352== Generated result file: /build/timeline.nvprof
* Running nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==447== NVPROF is profiling process 447, command: python m3.1.py 100
Loading model... done
New Inference
==447== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)" (done)
Op Time: 1.260088l events
Op Time: 0.001198
Correctness: 0.69 Model: ece408
==447== Generated result file: /build/forward1_analysis.nvprof
```

Figure 7: Demo of Nvprof forward1 analysis

2.6 NVVP analysis & observations

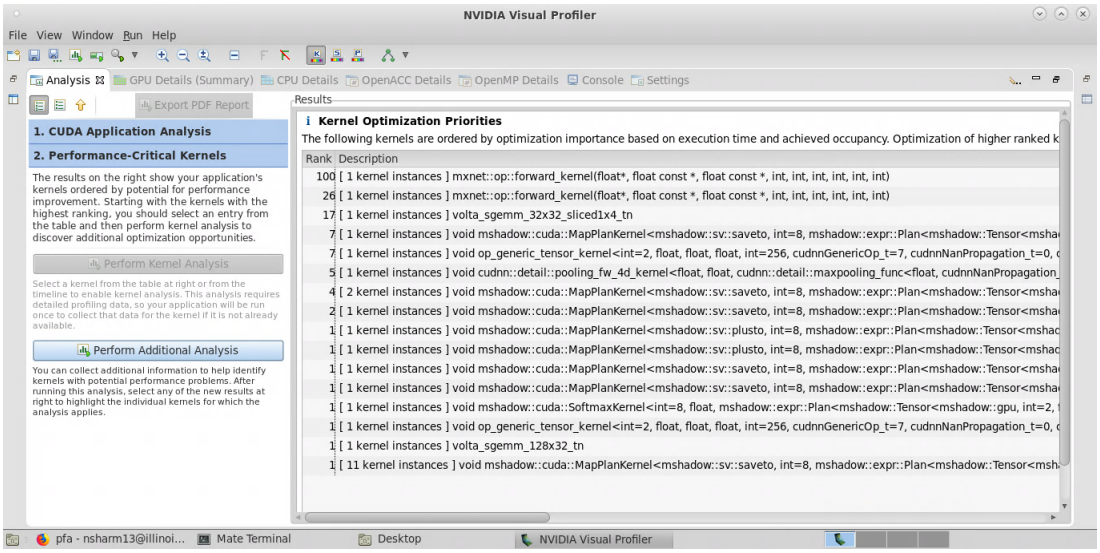

Figure 8: Kernel Usage

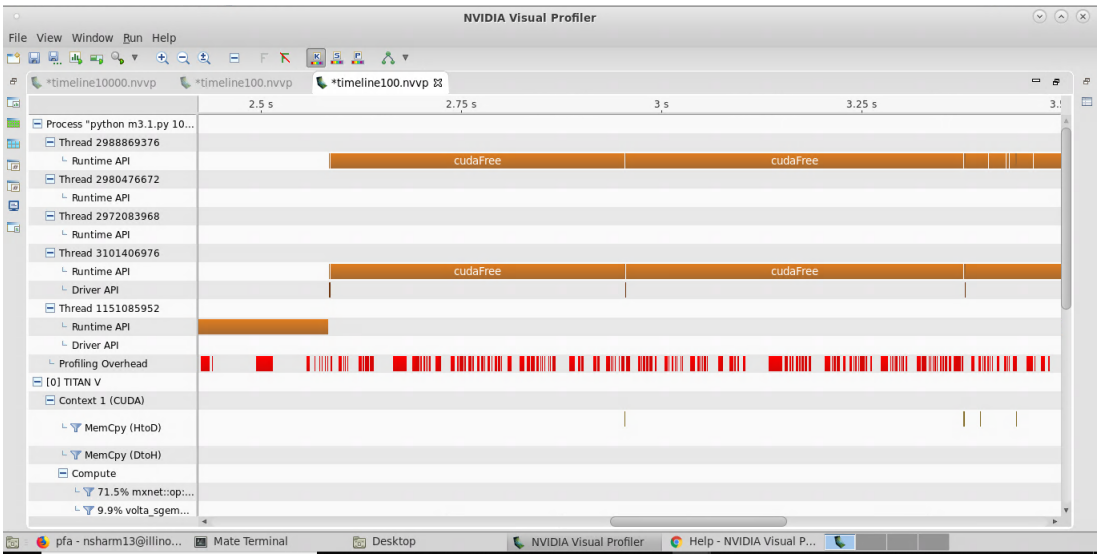

Figure 9: Trace for dataset size 100

Analysis Report

mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)

Duration	274.046 µs
Grid Size	[100,12,81]
Block Size	[8,8,1]
Registers/Thread	40
Shared Memory/Block	676 B
Shared Memory Executed	32 KiB
Shared Memory Bank Size	4 B

(0) TITAN V

GPU UUID	GPU-01e20778-ad5-31e2-acb7-80d0b9d6c701
Compute Capability	7.0
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	655 36
Max. Registers per Multiprocessor	655 36
Max. Grid Dimensions	[2147483 3647, 655 35, 655 35]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	29.798 TeraFLOP/s
Single Precision FLOP/s	14.899 TeraFLOP/s
Double Precision FLOP/s	7.45 TeraFLOP/s
Number of Multiprocessors	80
Multiprocessor Clock Rate	1.455 GHz
Concurrent Kernel	true
Max IPC	4
Threads per Warp	32
Global Memory Bandwidth	652.8 GiB/s
Global Memory Size	11.755 GiB
Constant Memory Size	64 KiB
L2 Cache Size	4.5 MiB
Memcpy Engines	7
PCIe Generation	3
PCIe Link Rate	8 Gbit/s
PCIe Link Width	8

Figure 10: Analysis generated by NVVP for dataset size 100

We used shared memory to optimise the kernel code. This reduced traffic to global memory. The use of shared memory tiling results in a very high level of acceleration in the execution of the kernel.

3 Milestone 4

The three optimizations are:

- Weight matrix in constant memory
- Sweeping various parameters to find best values (block sizes, amount of thread coarsening)
- Unroll + shared-memory Matrix multiply

3.1 Weight matrix in constant memory

The weight matrix is loaded into constant memory, that is

$$M * C * K * K$$

size. This optimization was chosen because all the images in the mini batch use the same M*C filter banks. Accessing all elements for constant memory reduces global memory traffic, thus giving us an increase in performance as can be observed from below.

3.1.1 Code

- Host code

```
template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float> &w)
{
    // Extract the tensor dimensions into B,M,C,H,W,K
    const int B = x.shape_[0];
    const int M = y.shape_[1];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = w.shape_[3];
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    int W_grid = W_out/TILE_WIDTH; // number of horizontal tiles per output map
    if (W_out % TILE_WIDTH) W_grid++;
    int H_grid = H_out/TILE_WIDTH; // number of vertical tiles per output map
    if (H_out % TILE_WIDTH) H_grid++;
    int Z = H_grid * W_grid;

    cudaMemcpyToSymbol(W_Const, w.dptr_ , M*C*K*K*sizeof(float), 0, cudaMemcpyDeviceToDevice);
    // Set the kernel dimensions
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 gridDim(B,M,Z);
    //X,Y,Z=> no. of batches, no. of output features, linearized blocks of each output feature


    forward_kernel<<< gridDim, blockDim >>>(y.dptr_,x.dptr_,w.dptr_,B,M,C,H,W,K, W_grid);
    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}
```

Figure 11: Host code

- Kernel code

```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

namespace mxnet
{
namespace op
{

#define TILE_WIDTH 16
#define WDIM 60*1024 / sizeof(float)
#define CH 3


__constant__ float W_Const[WDIM];

__global__ void forward_kernel(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, int W_grid)
{

#define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
#define w4d(i3, i2, i1, i0) W_Const[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

int b, m, c, h, w, p, q, h0, w0, h_base, w_base;
const int H_out = H - K + 1;
const int W_out = W - K + 1;

b = blockIdx.x; //b is no. of batches
m = blockIdx.y; //m is no. of output features

h_base = ((blockIdx.z / W_grid)*TILE_WIDTH);
h0 = threadIdx.y;
w_base = ((blockIdx.z % W_grid)*TILE_WIDTH);
w0 = threadIdx.x;
h = h_base + h0;
w = w_base + w0;

float sum = 0.0f;

if(h < H_out && w < W_out){
    for(c = 0; c < C; c++){
        for(p =0; p<K;p++){
            for(q = 0; q <K;  q++){
                sum += x4d(b,c,h+p,w+q) * w4d(m,c,p,q);
            }
        }

    }
    y4d(b,m,h,w) = sum;
}

#undef y4d
#undef x4d
#undef k4d
    }
```

Figure 12: Kernel

3.1.2 Dataset 100

- Op Time for layer 1: 1.041381s (for no optimization, time: 0.000280s)
- Op Time for layer 2: 0.000990s (for no optimization, time: 0.000934s)
- Correctness: 0.76

3.1.3 Dataset 1000

- Op Time for layer 1: 0.002739s (for no optimization, time: 0.003020s)
- Op Time for layer 2: 0.008308s (for no optimization, time: 0.009929s)
- Correctness: 0.767

3.1.4 Dataset 10000

- Op Time for layer 1: 0.027288s (for no optimization, time: 0.027775s)
- Op Time for layer 2: 0.087712s (for no optimization, time: 0.088688s)
- Correctness: 0.7653

3.1.5 Analysis using NVVP

- Analysis of compute, bandwidth and latency: As it can be noticed from the figure below, the kernel performance is bound by the compute which is utilized mostly by arithmetic operations. There is stark improvement in the operation times as can be seen from the time recorded above. For data set of 100 images, there is increase in time for first layer due to control divergence.

Figure 13: Analysis for Weight matrix in constant memory part 1

- Analysis of compute resources: There is control divergence of 16.5% and warp execution efficiency is 84.8%.

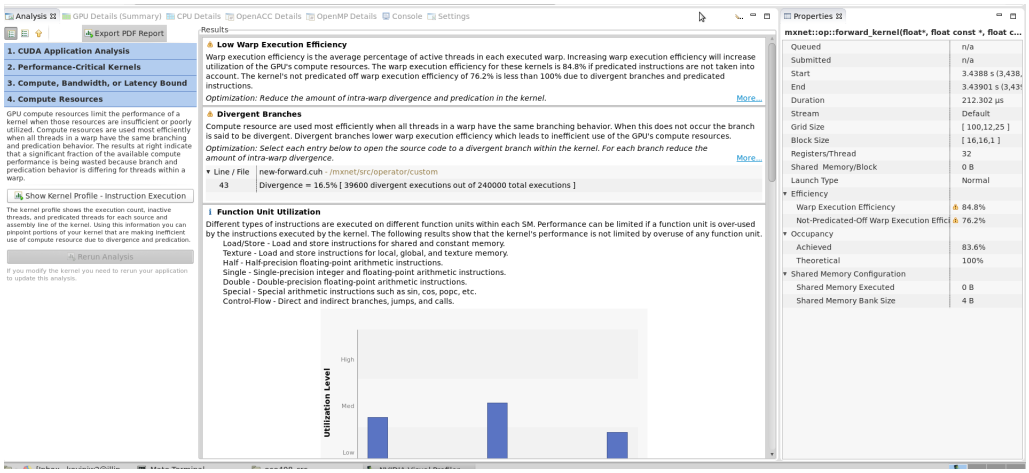

Figure 14: Analysis for Weight matrix in constant memory part 2

3.2 Sweeping various parameters to find best values (block sizes, amount of thread coarsening

This optimization is for choosing the best suited values for TILE_WIDTH and TILE_HEIGHT.

3.2.1 Code

- Host code

```
//Host code
template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float> &k)
{
    // Extract the tensor dimensions into B,M,C,H,W,K
        const int B = x.shape_[0];      // Batch size
    const int M = y.shape_[1];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = k.shape_[3];
        const int H_out = H - K + 1;
        const int W_out = W - K + 1;

    // Determine optimal block dimensions
        int block_width, block_height;
        for (int i = 16; i > 2; i--) {
                block_height = i;
                if (H_out % block_height == 0) {
                        break;
                }
        }
        block_width = 32;

        int W_grid = W_out / block_width;
        int H_grid = H_out / block_height;
        if (W_out % block_width) W_grid++;
        if (H_out % block_height) H_grid++;
        int Z = H_grid * W_grid;
        dim3 blockDim(block_width, block_height, 1);
        dim3 gridDim(B,M,Z);

    // Call the kernel
    forward_kernel<<<gridDim, blockDim>>>(y.dptr_, x.dptr_, k.dptr_, B,M,C,H,W,K);
    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
}
```

Figure 15: Host code

- Kernel code

```
__global__ void forward_kernel(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)
{
#define BLOCK_WIDTH blockDim.x
#define BLOCK_HEIGHT blockDim.y
// helpful macros for indexing
#define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

// ***Very basic convolution implementation, DOES NOT use shared memory tiling
const int H_out = H - K + 1;
const int W_out = W - K + 1;
int W_grid = W_out / BLOCK_WIDTH;
if (W_out % BLOCK_WIDTH) W_grid++;
int H_grid = H_out / BLOCK_HEIGHT;
if (H_out % BLOCK_HEIGHT) H_grid++;
int n, m, h, w;
n = blockIdx.x;
m = blockIdx.y;
h = (blockIdx.z / W_grid)*BLOCK_HEIGHT + threadIdx.y;
w = (blockIdx.z % W_grid)*BLOCK_WIDTH + threadIdx.x;

if(h < H_out && w < W_out) {
        float sum = 0.0;
        for (int c = 0; c < C; c++) {
                for (int p = 0; p < K; p++) {
                        for (int q = 0; q < K; q++) {
                                sum += x4d(n, c, h + p, w + q) * k4d(m, c, p, q);
                        }
                }
        }
        y4d(n, m, h, w) = sum;
}
```

Figure 16: Kernel

3.2.2 Dataset 100

- Op Time for layer 1: 1.272478s
- Op Time for layer 2: 0.001017s
- Correctness: 0.76

3.2.3 Dataset 1000

- Op Time for layer 1: 0.003231s
- Op Time for layer 2: 0.010110s
- Correctness: 0.767

3.2.4 Dataset 10000

- Op Time for layer 1: 0.031941s
- Op Time for layer 2: 0.099054s
- Correctness: 0.7653

3.2.5 Analysis using NVVP

- Analysis of compute, bandwidth and latency: The control divergence is 33.3% and warp execution efficiency is 73.8%.

Figure 17: Analysis for different parameters part 1

- Analysis of compute resources

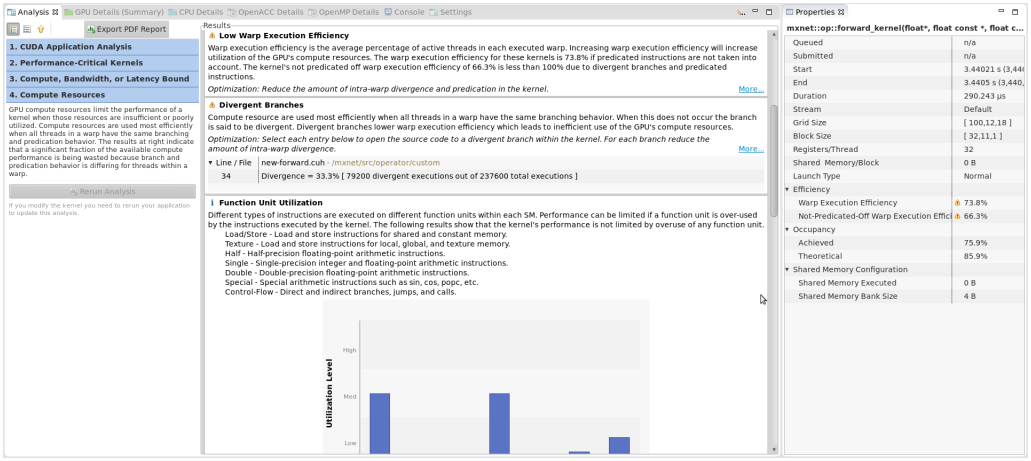

Figure 18: Analysis for different parameters part 1

3.3 Unroll + shared-memory Matrix multiply

When we perform unrolling, we essentially club together the elements of all the feature maps by duplicating all the elements that we will require to compute one output element, and storing them into a combined matrix. Additionally, we also concatenate all the weight matrices into a combined matrix. Hence, unrolling and shared memory is an extremely effective method for optimizing the process of convolution.

3.3.1 Code

- Host code

```
template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float> &k)
{

    // Use mxnet's CHECK_EQ to do assertions.
    // Remove this assertion when you do your implementation!
    // CHECK_EQ(0, 1) << "Remove this line and replace with your implementation";

    // Extract the tensor dimensions into B,M,C,H,W,K
    // ...
    const int B = x.shape_[0];
    const int M = y.shape_[1];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = k.shape_[3];
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    float* Y = y.dptr_;
    float* X = x.dptr_;
    cudaMemcpyToSymbol(Kernel, k.dptr_, sizeof(float)*M*C*K*K);


    float* X_unrolled;
    int size = C*K*K*H_out*W_out;
    cudaMalloc(&X_unrolled, sizeof(float)*size);
    for (int b = B; b--; ) {
        unroll(X_unrolled, size, X+b*C*H*W, C, K, H, W);
        gemm(X_unrolled,  Y+b*M*H_out*W_out,  C*K*K,  M,  H_out*W_out);
    }
    cudaFree(X_unrolled);

    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}
```

Figure 19: Host code

- Kernel code

```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#define BLOCK_SIZE 1024
#define TILE_WIDTH 32
#define KERNEL_SIZE 14112

#include <mxnet/base.h>

namespace mxnet
{
namespace op
{


__constant__ float Kernel[KERNEL_SIZE];

__global__ void matrixMultiplyShared(float *B, float *C, int numAColumns, int numCRows, int numCColumns)
{
  // numARows = numCRows
  // numBRows = numAColumns
  // numBColumns = numCColumns

  //@@ Insert code to implement matrix multiplication here
  //@@ You have to use shared memory for this MP
  // __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
  __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];
  int rowIx = blockIdx.y * blockDim.y + threadIdx.y;
  int colIx = blockIdx.x * blockDim.x + threadIdx.x;

  float result = 0;

  for (int tileIx = 0; tileIx < ceil(1.0*numAColumns/TILE_WIDTH); tileIx++) {

    // int col = tileIx*TILE_WIDTH+threadIdx.x;
    // if (col < numAColumns)
    //   tileA[threadIdx.y][threadIdx.x] = Kernel[rowIx*numAColumns+col];
    // else
    //   tileA[threadIdx.y][threadIdx.x] = 0;


    int row = tileIx*TILE_WIDTH+threadIdx.y;
    if (row < numAColumns)
      tileB[threadIdx.y][threadIdx.x] = B[row*numCColumns + colIx];
    else
      tileB[threadIdx.y][threadIdx.x] = 0;


    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; k++)
      if (tileIx*TILE_WIDTH+k < numAColumns)
        result += Kernel[rowIx*numAColumns+tileIx*TILE_WIDTH+k]*tileB[k][threadIdx.x];

    __syncthreads();
  }

  if ((rowIx < numCRows) && (colIx < numCColumns)) {
    C[rowIx*numCColumns+colIx] = result;
  }

}
```

Figure 20: Kernel Part 1

```
void gemm(float* X_unrolled,  float* Y, int CKK, int M, int HW) {
    // matrixMultiplyShared(float *A, float *B, float *C,
    //                                 int numAColumns, int numCRows, int numCColumns)
    // W_unroll = K
    int blockDimX = TILE_WIDTH, blockDimY = TILE_WIDTH;
    int gridDimY = ceil(1.0*M/blockDimY), gridDimX = ceil(1.0*HW/blockDimX);
    dim3 gridDim (gridDimX, gridDimY), blockDim (blockDimX, blockDimY);
    matrixMultiplyShared<<<gridDim, blockDim>>>(X_unrolled, Y, CKK, M, HW);
}


__global__ void unrollKernel(float* X_unrolled, int size, float* X, int C, int K, int H, int W) {
    int ix = blockDim.x*blockIdx.x + threadIdx.x;
    if (ix >= size)
        return;
    int H_out = H-K+1;
    int W_out = W-K+1;
    int row = ix/(H_out*W_out);
    int col = ix%(H_out*W_out);
    int q = row % K;
    row /= K;
    int p = row % K;
    int c = row / K;
    int w = col % W_out;
    int h = col / W_out;
    X_unrolled[ix] = X[(c) * (H * W) + (h+p) * (W) + w+q];
}

void unroll(float* X_unrolled, int size, float* X, int C, int K, int H, int W) {
    int gridDim = ceil(1.0*size/BLOCK_SIZE);
    unrollKernel<<<gridDim, BLOCK_SIZE>>>(X_unrolled, size, X, C, K, H, W);
}
```

Figure 21: Kernel Part 2

3.3.2 Dataset 100

- Op Time for layer 1: 0.001937s
- Op Time for layer 2: 0.004298s
- Correctness: 0.76

3.3.3 Dataset 1000

- Op Time for layer 1: 0.018220s
- Op Time for layer 2: 0.040287s
- Correctness: 0.767

3.3.4 Dataset 10000

- Op Time for layer 1: 0.030497s
- Op Time for layer 2: 0.086416s
- Correctness: 0.7653

3.3.5 Analysis using NVVP

- Analysis of compute, bandwidth and latency for matrix multiplication: This optimization transforms the forward operation of the convolution layer to the multiplication of the two aforementioned combined matrices. This decreases the compute times in comparison to the case without unrolling, in which each weight mask would be applied to each feature mask separately. Moreover, using shared memory tiling to perform multiplication decreases the global memory access times further resulting in increased performance.

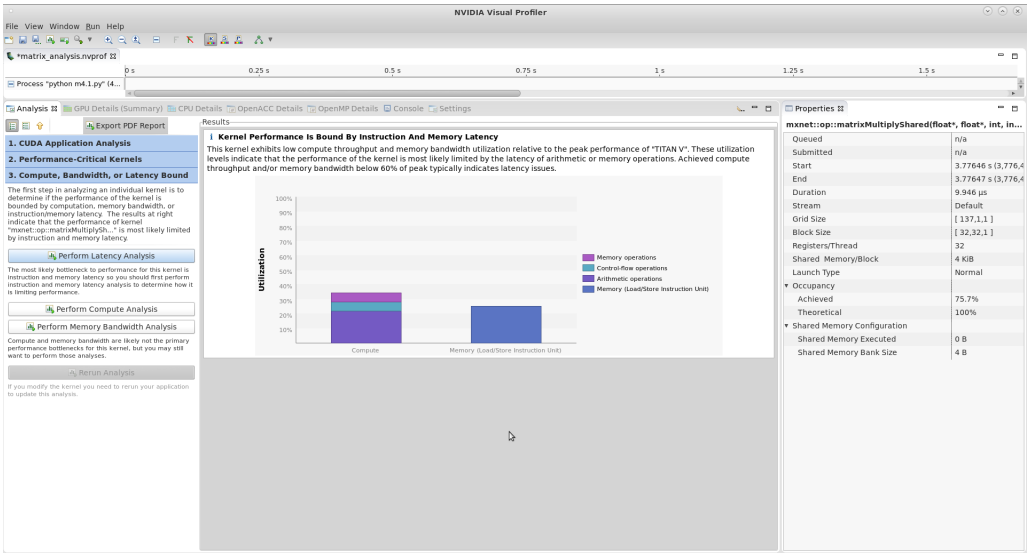

Figure 22: Analysis for unroll & shared matrix multiplication part 1

- Analysis of compute resources for matrix multiplication

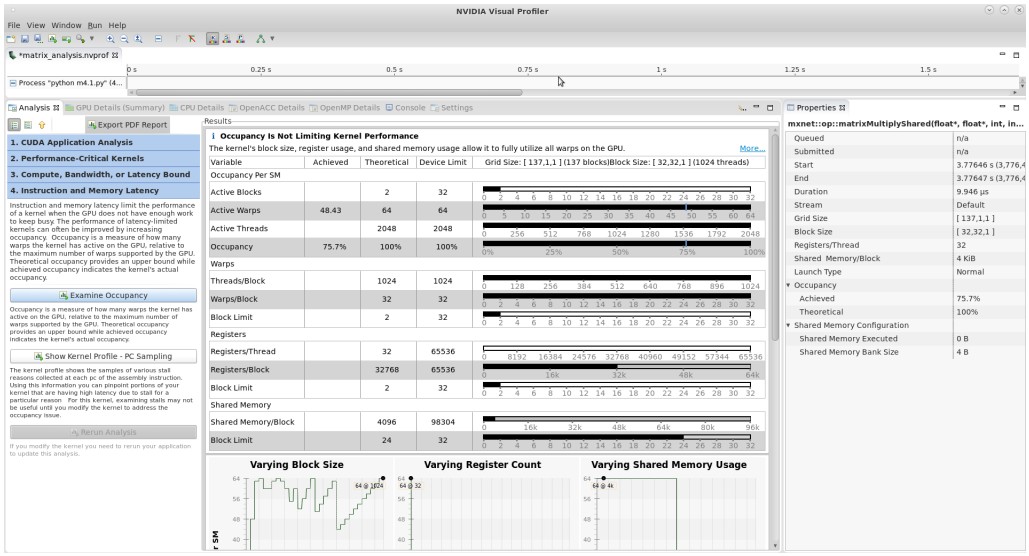

Figure 23: Analysis for unroll & shared matrix multiplication part 2

- Analysis of compute, bandwidth and latency for loop unroll

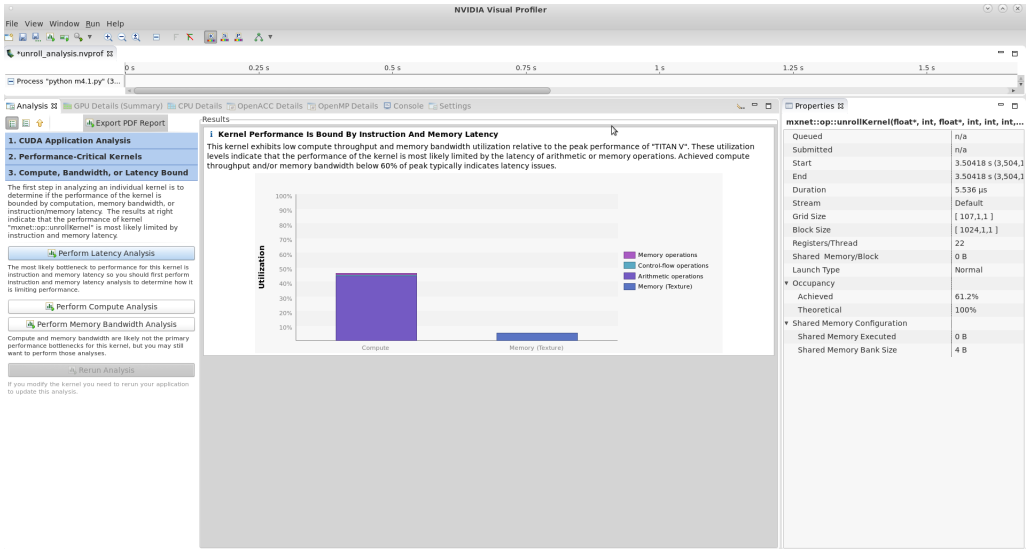

Figure 24: Analysis for unroll & shared matrix multiplication part 3

- Overall timeline for this optimization

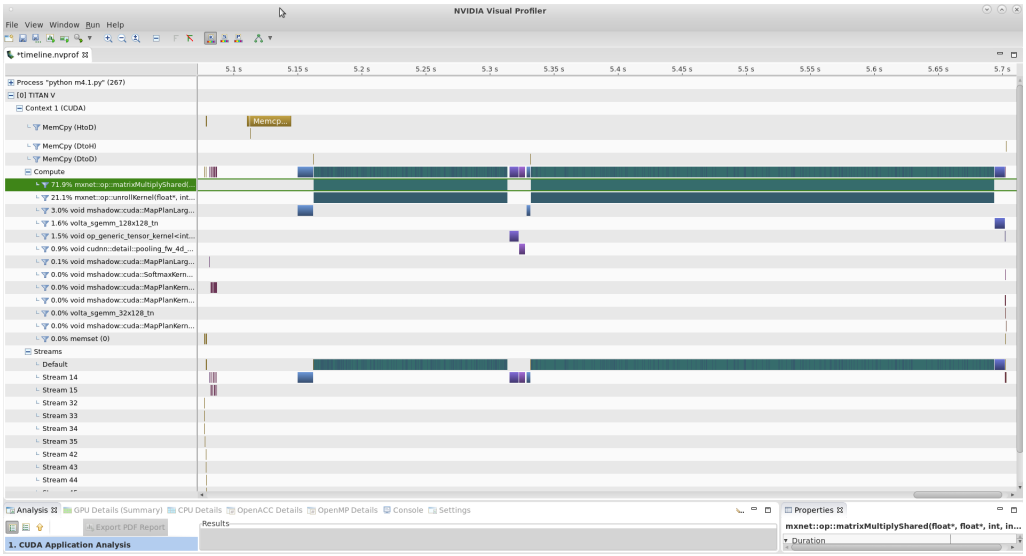

Figure 25: Timeline for unroll & shared matrix multiplication

4 Final

We have in all these 6 optimizations.

- Kernel fusion for unrolling and matrix-multiplication
- Shared Memory convolution
- Tuning with restrict and loop unrolling
- Unroll + shared-memory Matrix multiply
- Sweeping various parameters to find best values (block sizes)
- Weight matrix (kernel values) in constant memory

The last 3 are done in the previous milestone. We have combined Shared memory convolution and weights in constant memory as one optimization. We re-coded with grid dimensions to (linearized Y, M, B) to give us more speedup.

4.1 Naive implementation with new dimensions: Stats for comparison

4.1.1 Optimes

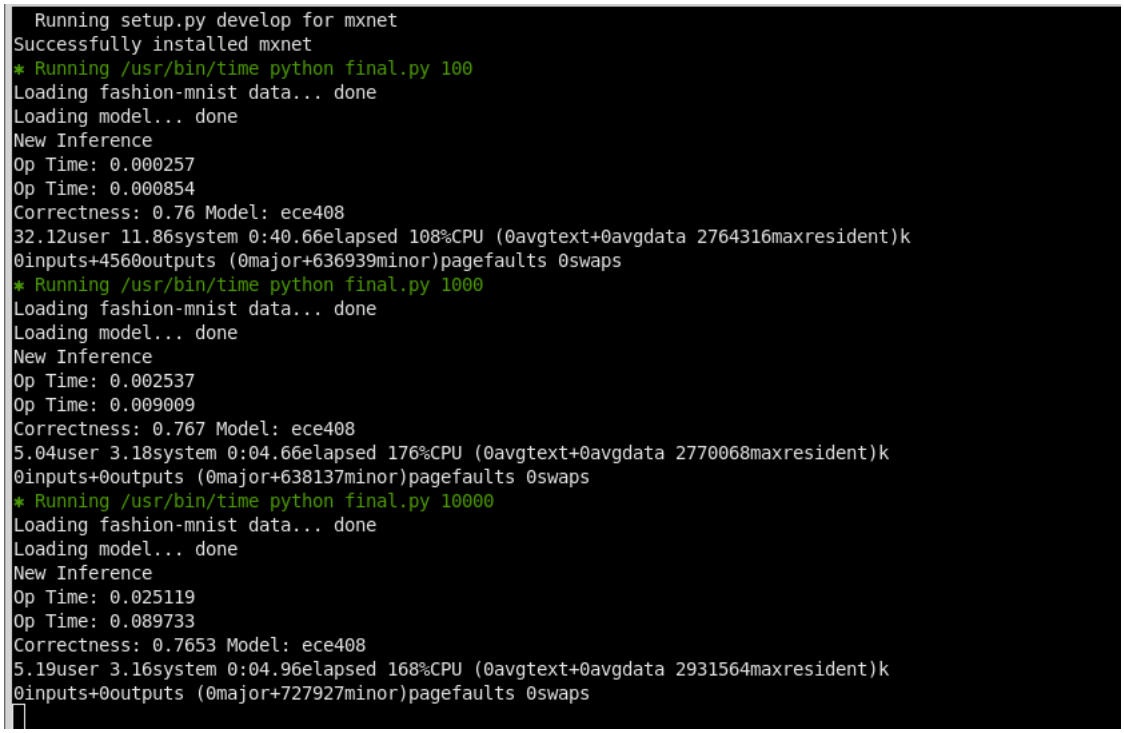

Figure 26: Naive implementation times

4.1.2 Analysis using NVVP

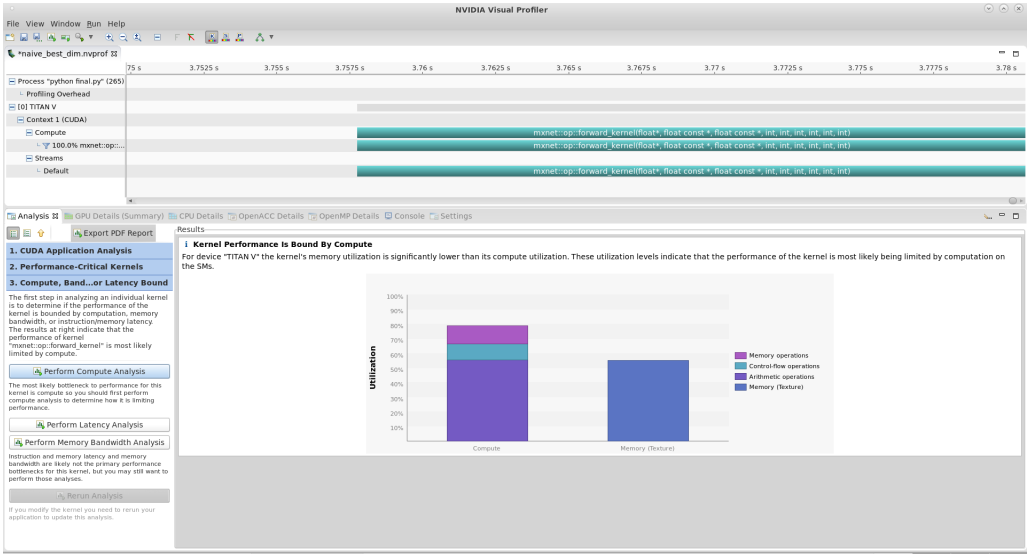

Figure 27: Analysis of Naive base part 1

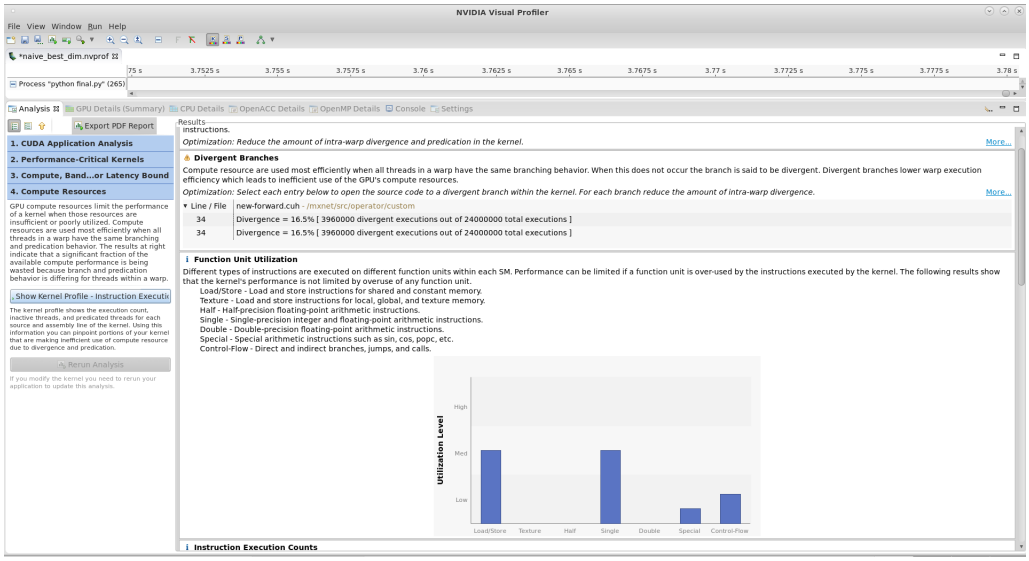

Figure 28: Analysis of Naive base part 2

4.2 Kernel fusion for unrolling and matrix-multiplication

The idea is to not call a separate kernel for unrolling but instead access the data indices without physically unrolling the input matrix. We did that using logical unrolling, that is, when loading elements for tiled matrix multiplication, we unrolled the input matrix and stored in the tile. Loading weights in constant memory and then storing it in tile did not help since, time to store weights in constant memory and then storing them back in shared memory caused the overhead.

This optimization struck us after we did separate kernels for unroll and matrix multiplication. This is our best optimization.

4.2.1 Optimes

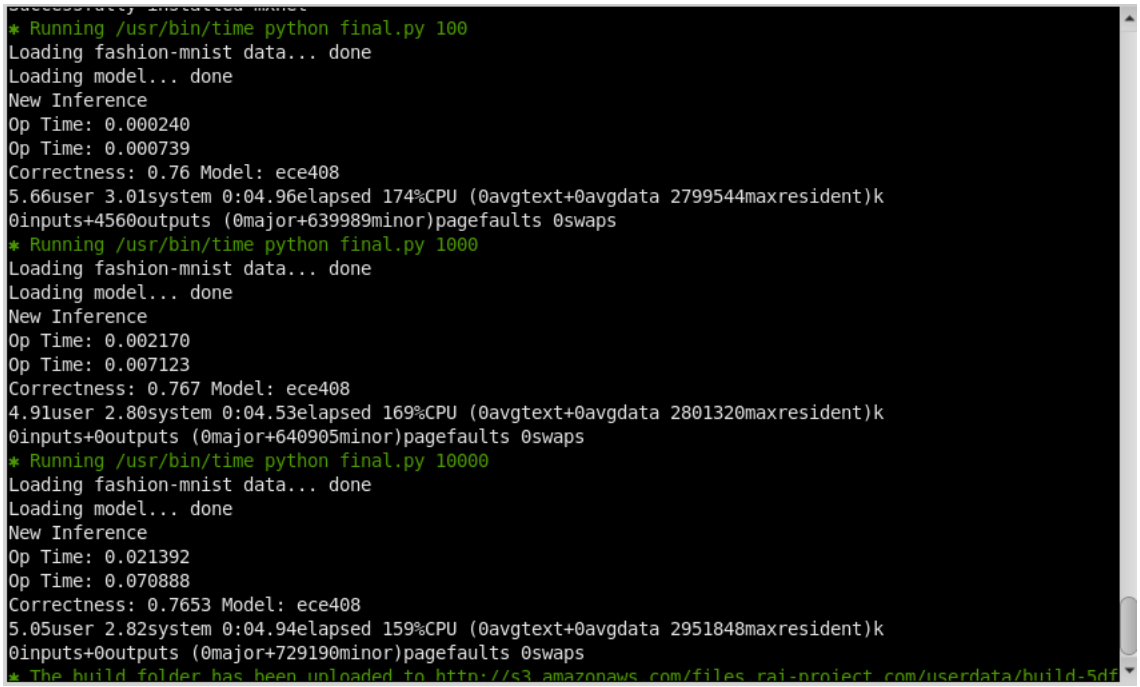

Figure 29: Kernel fusion unroll matrix multiplication times

4.2.2 Analysis using NVVP

The Kernel performance is good but limited by memory bandwidth and utilization of Shared memory is medium.

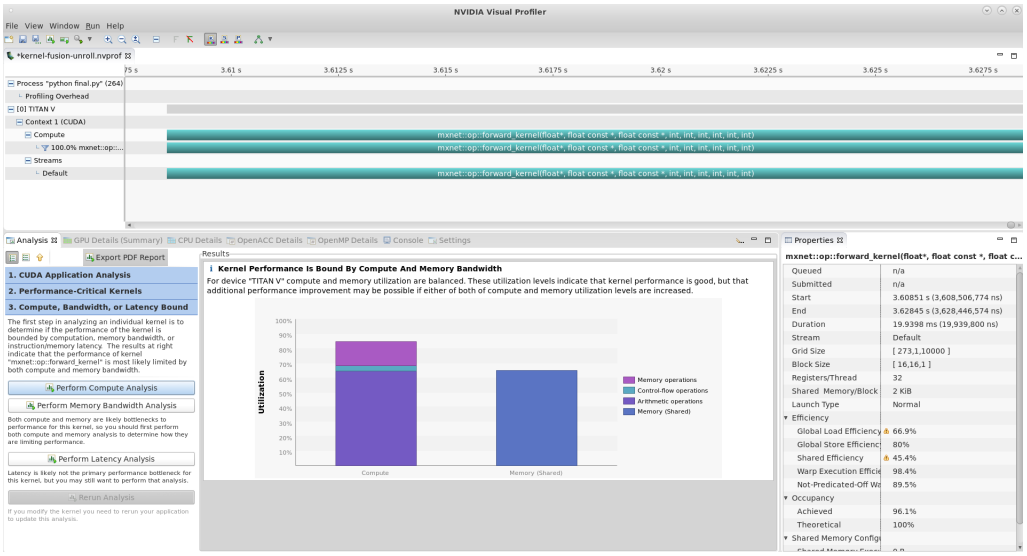

Figure 30: Analysis for kernel fusion unroll & shared matrix multiplication part 1

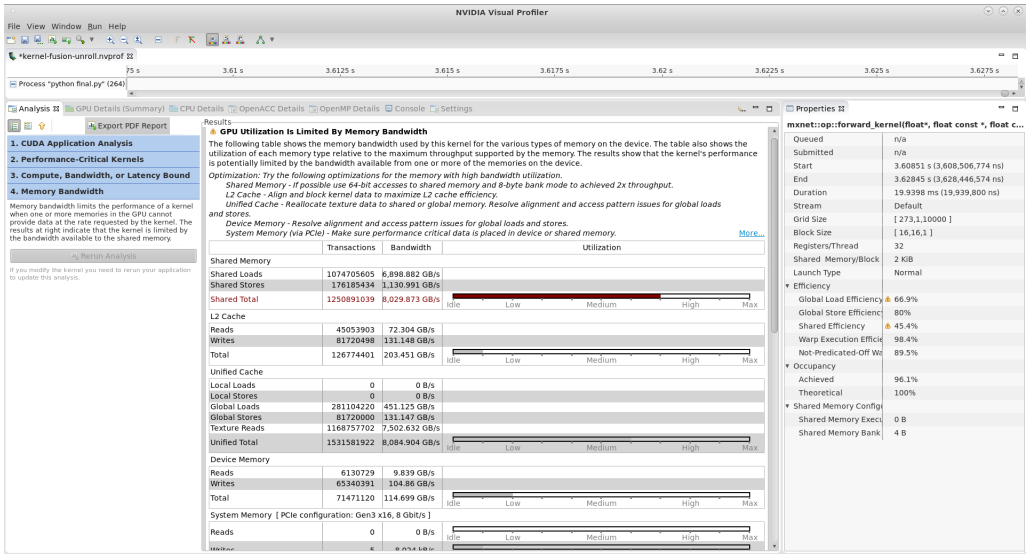

Figure 31: Analysis for kernel fusion unroll & shared matrix multiplication part 2

4.3 Shared memory convolution & weight matrix in Constant memory

We are implementing forward layer convolution of the CNN. The most apparent optimization was to load the input matrix (X) elements from Shared memory to reduce the global memory bottleneck. We implemented strategy 3 for convolution where only the BLOCK_SIZE is loaded in shared memory and halo cells are

accessed from global memory. We further combined this optimization with the weight matrix in constant memory to further increase performance.

4.3.1 Optimes

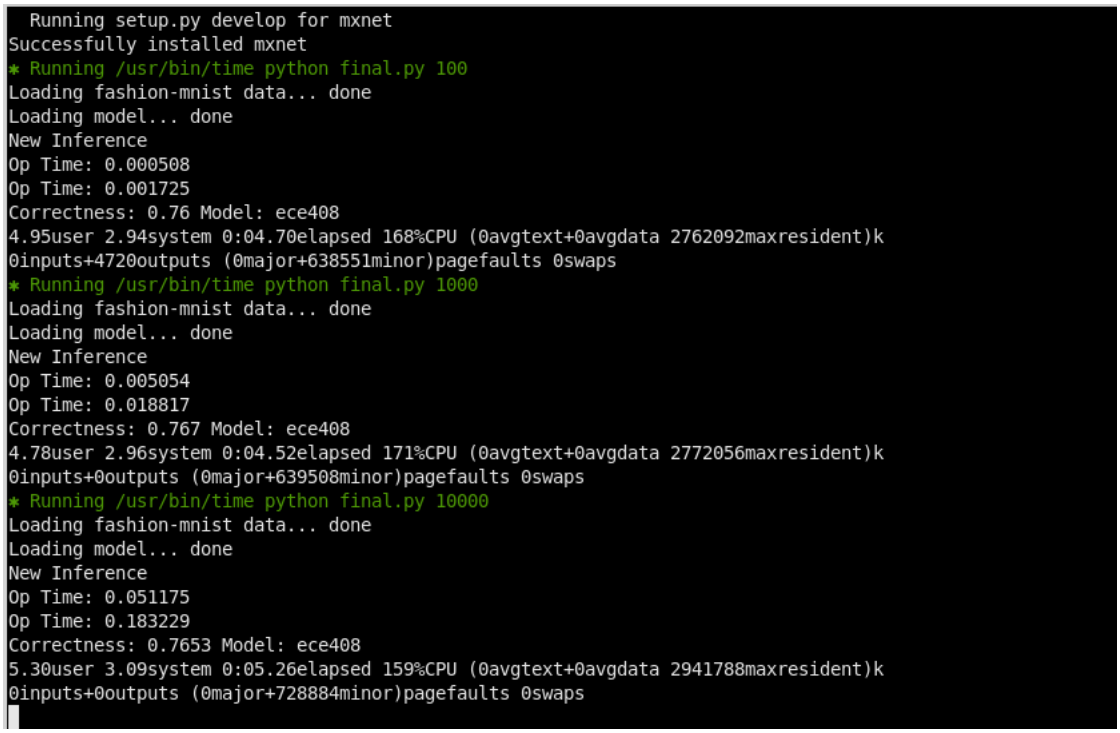

Figure 32: Shared memory convolution & weight matrix in Constant memory times

4.3.2 Analysis using NVVP

Here we noticed that memory utilization was significantly lower than compute utilization due to the limitation from computation on Streaming multiprocessors. There is observed to be a lot of intra-warp divergence and predication in the kernel due to divergent branch intructions.

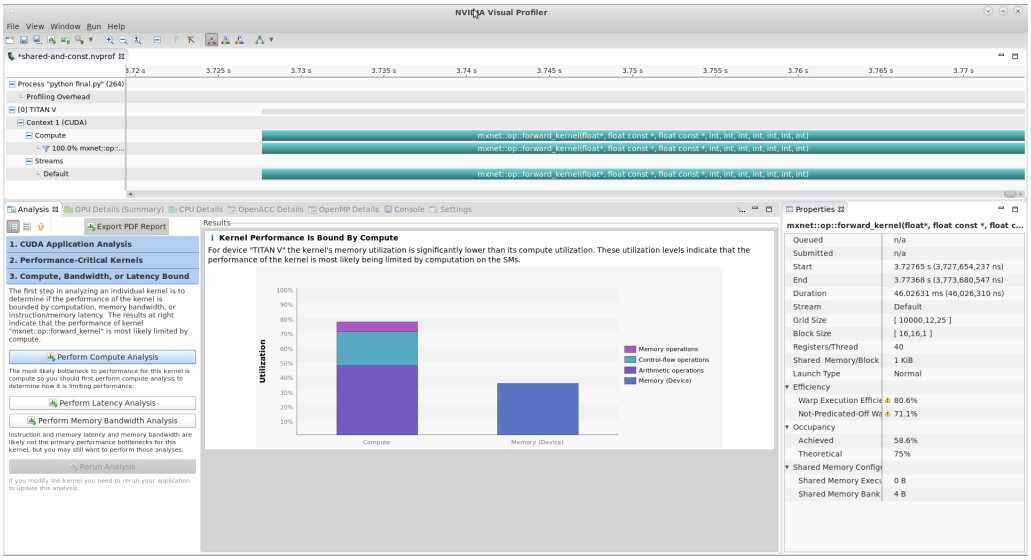

Figure 33: Analysis of Shared memory convolution & weight matrix in Constant memory part 1

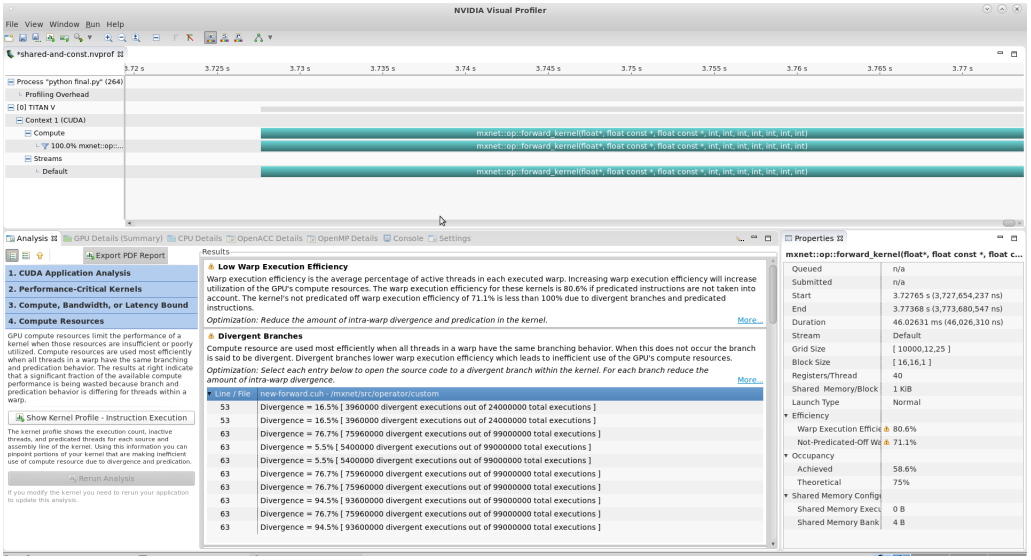

Figure 34: Analysis of Shared memory convolution & weight matrix in Constant memory part 2

4.4 Tuning with restrict and loop unrolling

Loop unrolling is a popular technique that attempts to optimize a program's execution speed at the expense of code size. We attempted this optimization after looking at the github page for the 408 project. We observed the following.

4.4.1 Optimes

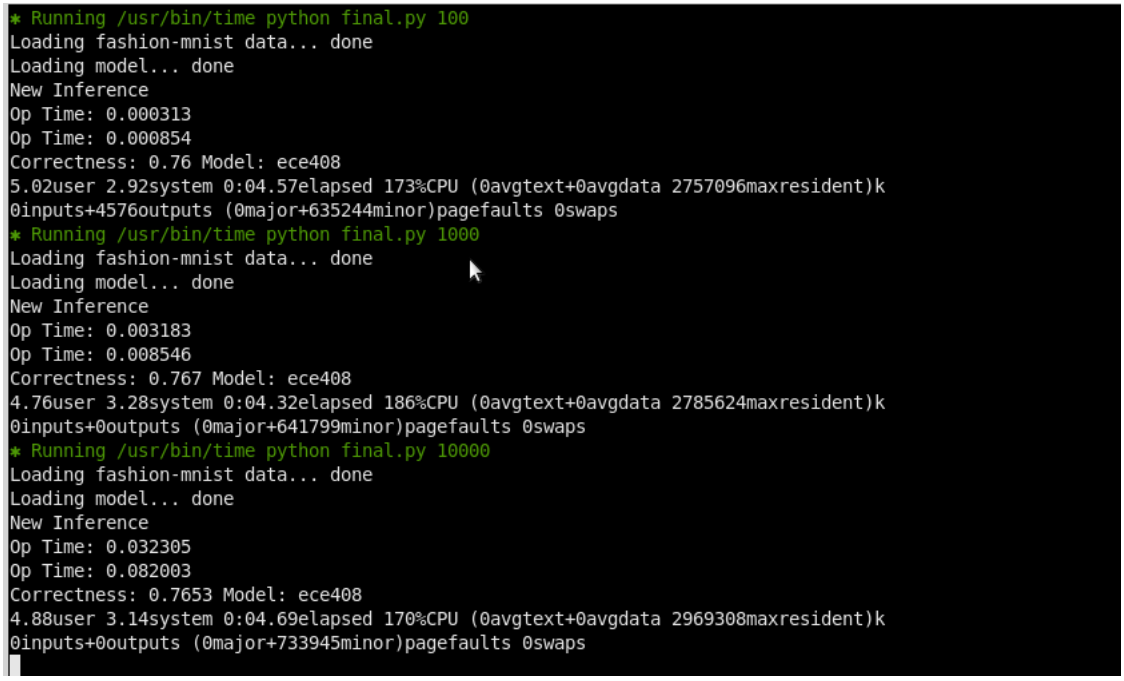

Figure 35: Tuning with restrict & loop unrolling times

4.4.2 Analysis using NVVP

We observed that performance is limited by the latency of arithmetic and memory operations. The kernel uses 40 register per thread, we have 10240 registers in use per block, this is increase in occupancy did not lead to increase in performance. However, we tried this to combine this optimization with kernel fusion, we did not notice any significant increase in speed.

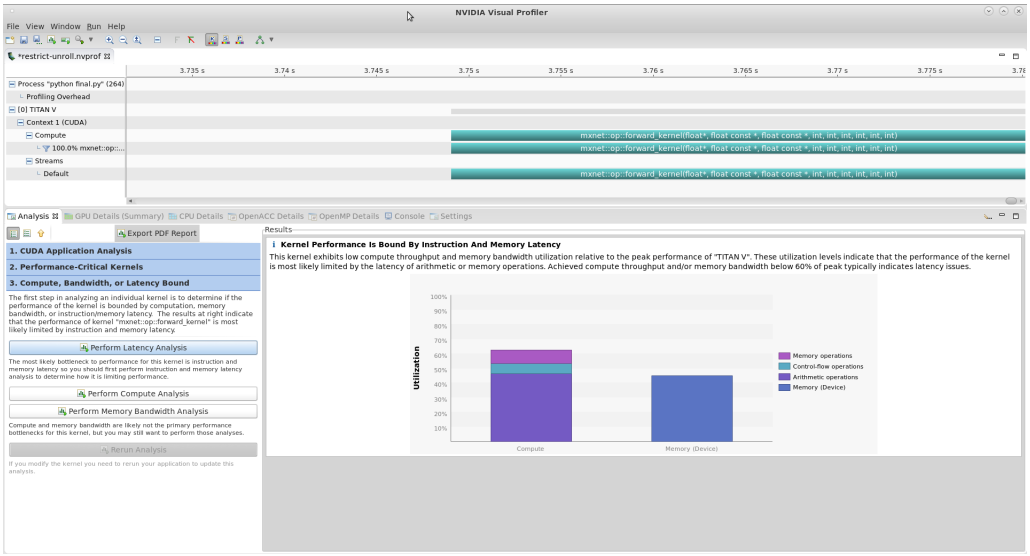

Figure 36: Analysis of Tuning with restrict & loop unrolling part 1

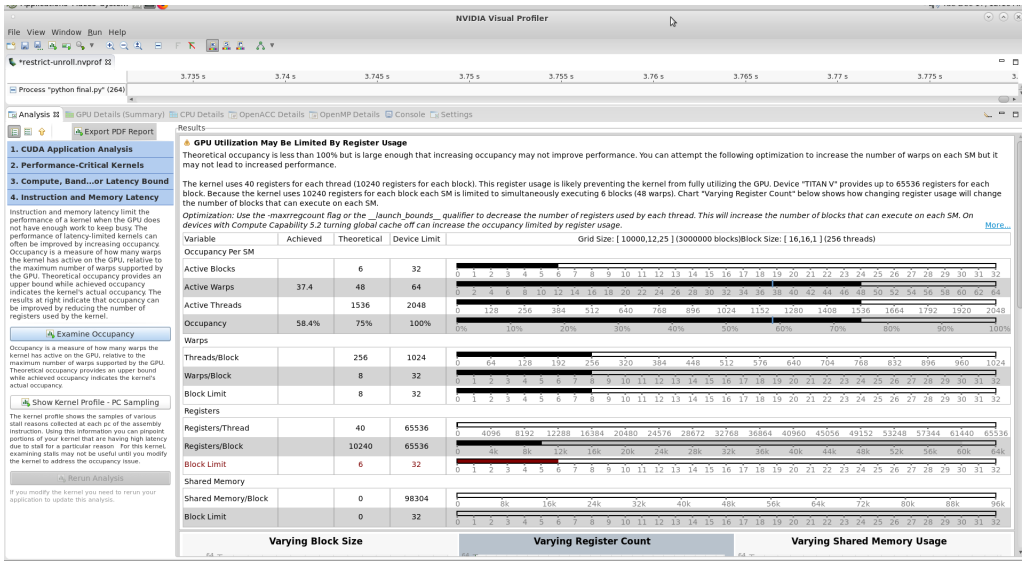

Figure 37: Analysis of Tuning with restrict & loop unrolling part 2

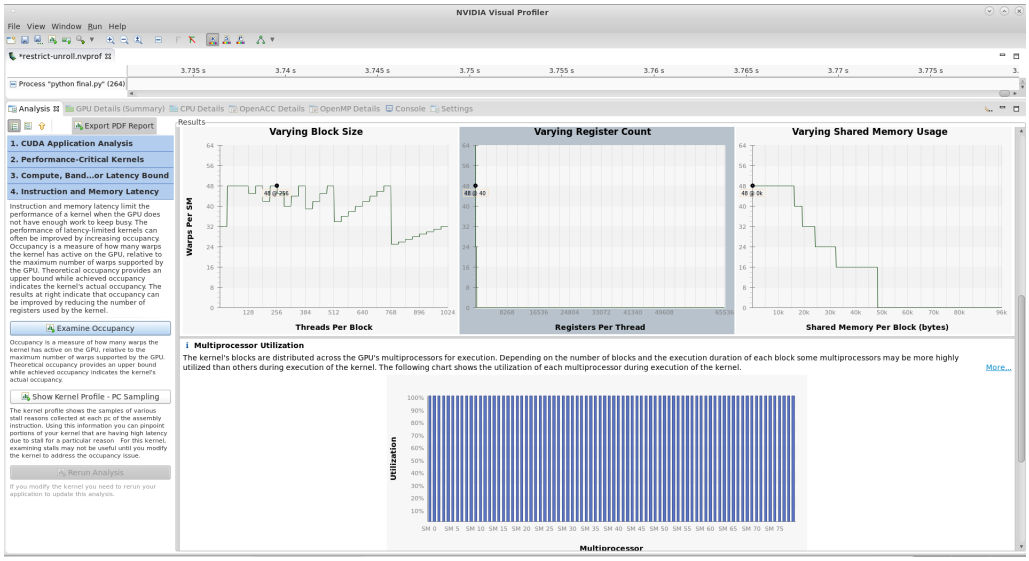

Figure 38: Analysis of Tuning with restrict & loop unrolling part 3

4.5 Parameter sweep with different dimensions

We noticed decrease in time for the (Y,M,B) dimensions. Here are the results.

4.5.1 Optimes

```
  Running setup.py develop for mxnet
Successfully installed mxnet
* Running /usr/bin/time python final.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000345
Op Time: 0.000910
Correctness: 0.76 Model: ece408
5.01user 2.91system 0:04.74elapsed 167%CPU (0avgtext+0avgdata 2768728maxresident)k
0inputs+4560outputs (0major+638726minor)pagefaults 0swaps
* Running /usr/bin/time python final.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.003096
Op Time: 0.008744
Correctness: 0.767 Model: ece408
4.76user 2.90system 0:04.53elapsed 169%CPU (0avgtext+0avgdata 2781436maxresident)k
0inputs+0outputs (0major+640986minor)pagefaults 0swaps
* Running /usr/bin/time python final.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.030652
Op Time: 0.084095
Correctness: 0.7653 Model: ece408
5.38user 3.27system 0:05.06elapsed 171%CPU (0avgtext+0avgdata 2942560maxresident)k
0inputs+0outputs (0major+730071minor)pagefaults 0swaps
```

Figure 39: Parameter sweep times

4.5.2 Analysis using NVVP

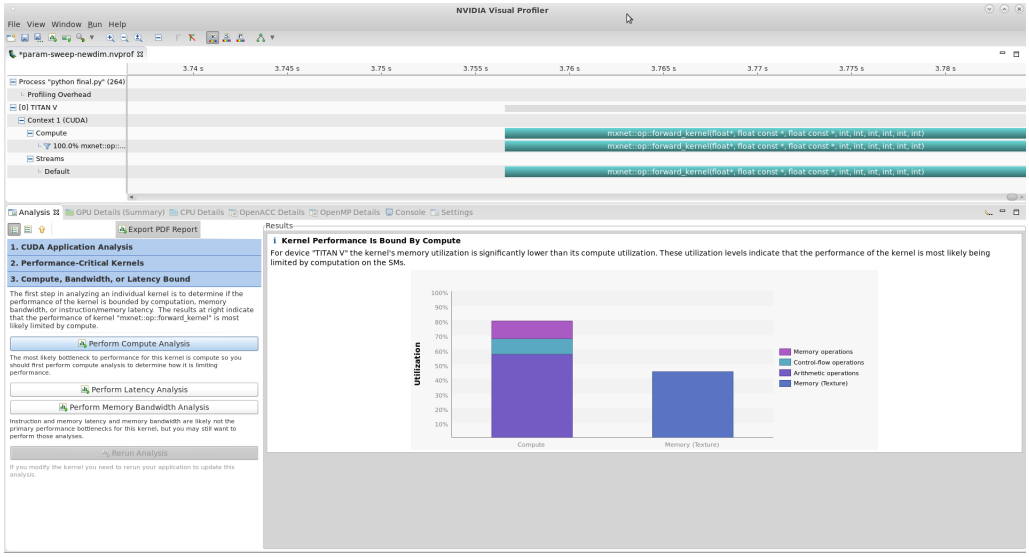

Figure 40: Analysis of Parameter sweeping part 1

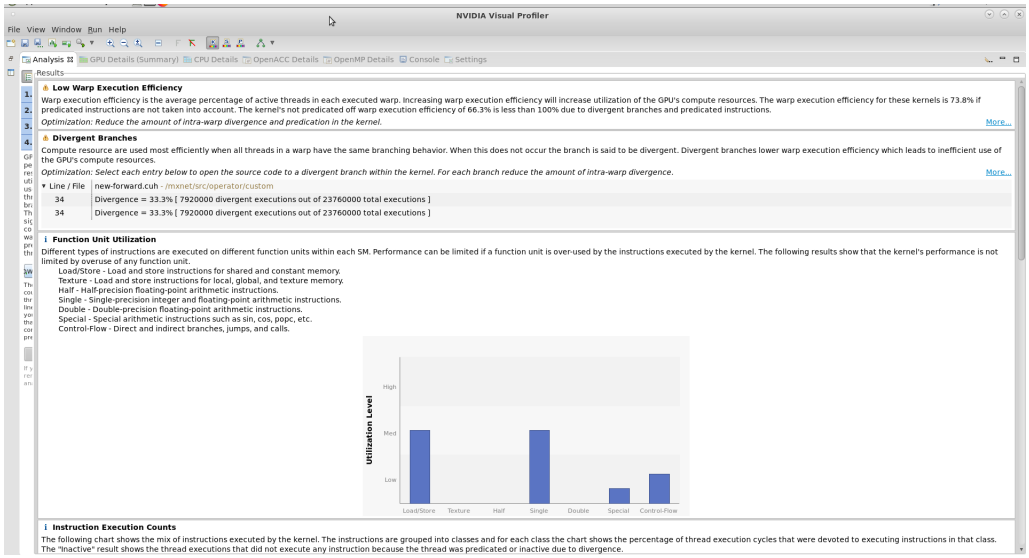

Figure 41: Analysis of Parameter sweeping part 2

4.6 Additional Optimizations

We tried these additional optimizations.

- Register tiling (Error in boundary checks)
- To use tensor cores (Tensor cores operate on half precision, not of much use)
- Kernel fusion + weights in constant memory (almost no effect)

4.7 Team Effort

We divided the optimizations as 2 per person and all of us equally contributed towards fetching the results and doing the report.