ECE408 Project Final Report

Shuyue Lai (shuyuel2), Yaxin Peng (yaxinp2), Jingyuan Zhang (jz61)

Team Name: tensor

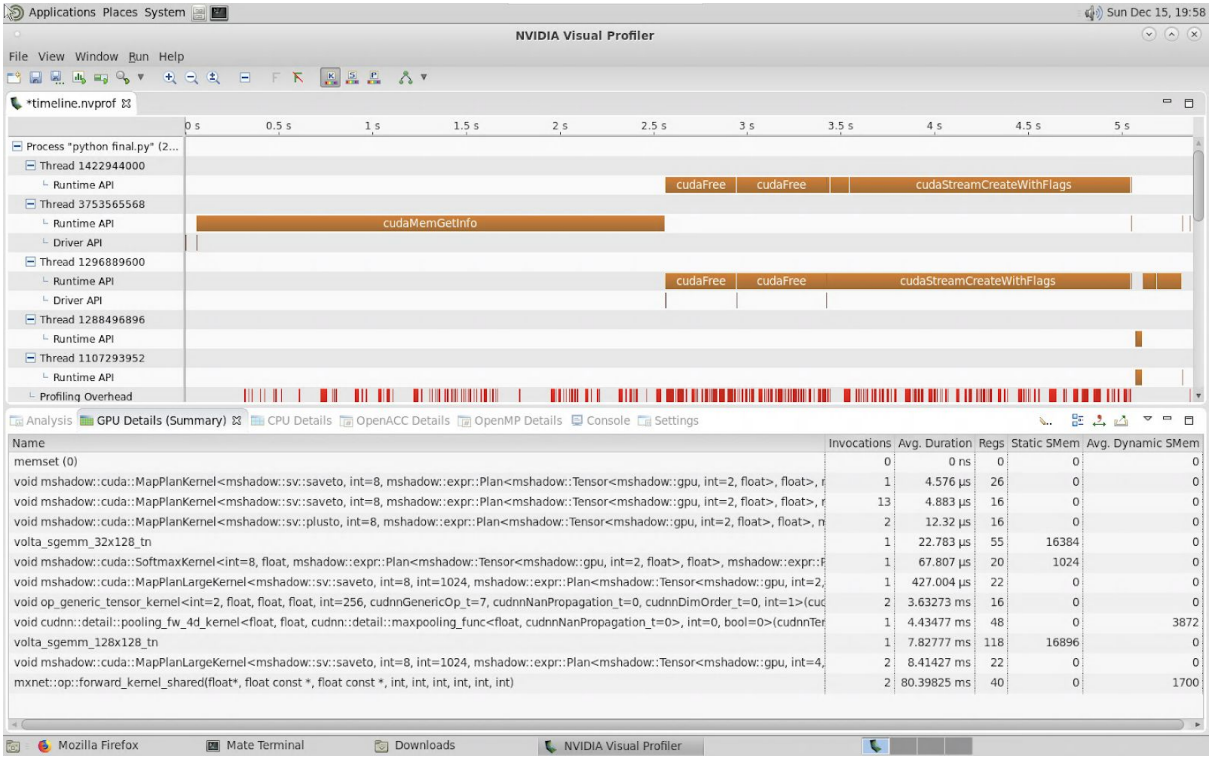
UIUC on Campus Students

Optimization 1: Shared Memory Convolution （result is in milestone 4)

Shared memory is shared by all threads in a thread block and has approximately lower latency and about 10x higher bandwidth than global device memory.[1]
By using Shared Memory Convolution, both input data X and kernel data W are stored into shared memory before computation. Since global memory is implemented with DRAM which is slow, to avoid global memory bottleneck, optimization 1 take advantage of shared memory which is smaller but fast by having several threads use the local version to reduce the memory bandwidth.
In this optimization, data is partitioned into subsets which fit into smaller but faster shared memory. Then subset from global memory will be loaded into shared memory and each data subset will be handled with one thread block.

[1] Kasia Swirydowicz (2018, February 26). Basic GPU optimization strategies. https://www.paranumal.com/single-post/2018/02/26/Basic-GPU-optimization-strategies

- Statistics and visualization on size = 10000:

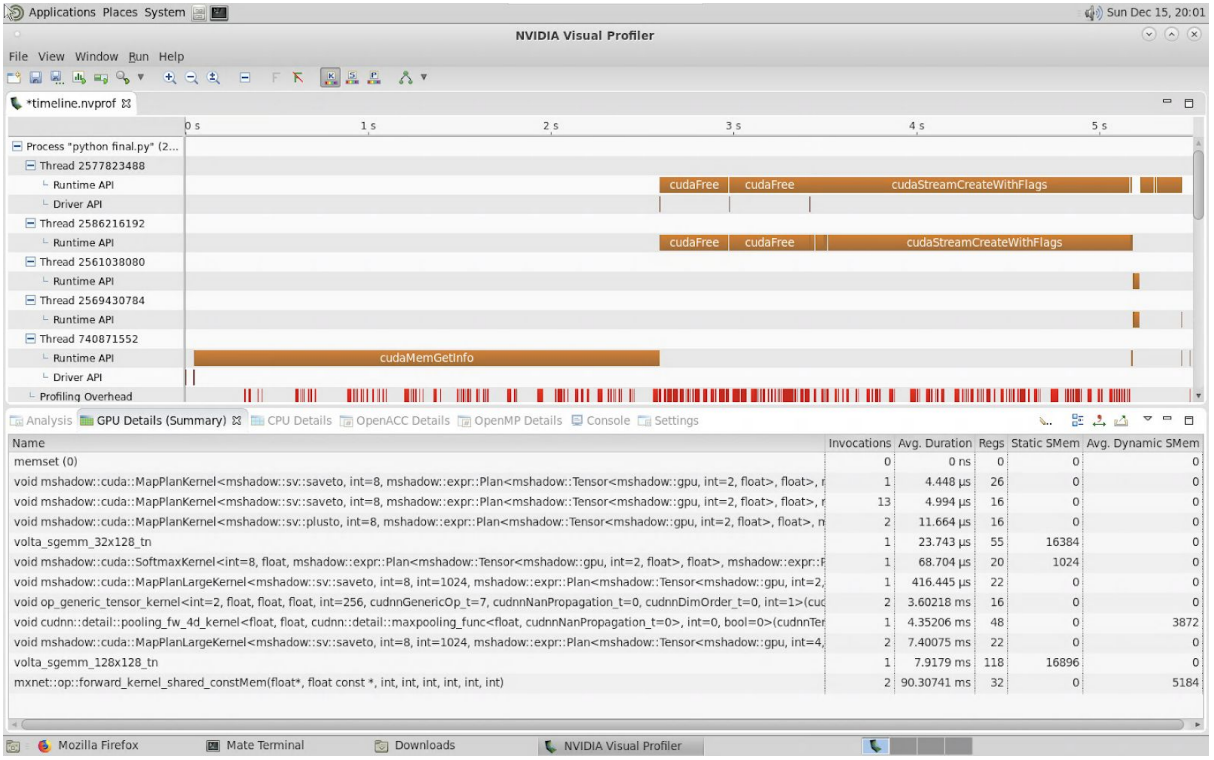


Optimization 2: Shared Memory with Weight Matrix (Kernel Values) in Constant Memory
（result is in milestone 4)

Different from optimization 1, kernel data W is stored into constant memory instead of shared memory in optimization 2. Constant memory is a read-only cache for global device memory. It is cached to constant cache and can be accessed by each thread. [2] Kernel data W in CNN remains same for each calculation. In addition, since constant memory is not particularly large and kernel data W is also not particularly large compared to input data X, using constant memory can improve performance.

[2] Kasia Swirydowicz (2018, February 26). Basic GPU optimization strategies. https://www.paranumal.com/single-post/2018/02/26/Basic-GPU-optimization-strategies

- Statistics and visualization on size = 10000:

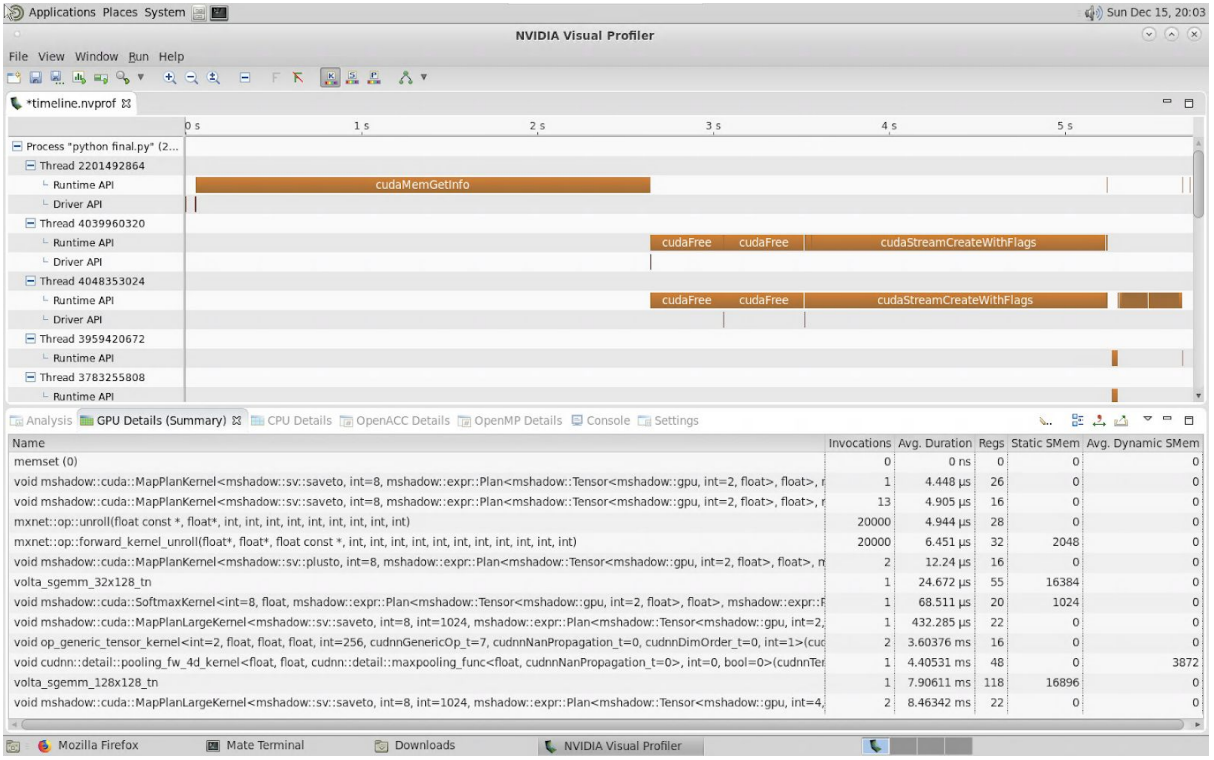


Optimization 3: Unroll + Shared-memory Matrix Multiply （result is in milestone 4)

Both input Data X and kernel are stored in shared memory before computation and will be modified as unroll matrix to do matrix multiplication. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve performance. But it also increases code size in the loop body which may increase pressure on register allocation, cause register spilling. Therefore, for unrolling, evaluation of tradeoffs is needed.[3]

[3] IGM. Unroll Nounroll. https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcux01/unroll.htm

- Statistics and visualization on size = 10000:



Optimization 4: Kernel Fusion for Unrolling and Matrix-Multiplication

As optimization 3, unrolling requires extra operations and memory resources to do index mapping and store new unrolling matrix. Performance can be improved use only the tiled matrix multiplication kernel without actual unrolling operation. Perform unrolling when loading the tile into shared memory by correctly calculating the data indices instead of having a seperate unrolling kernel. Fused kernel does both operation instead of 2 global memory load operations and 2 global memory store operations. By doing this, the savings can be significant for memory-bound operations on GPU since there is no need to do data transfer from original data to new unrolling matrix. Reducing redundant loads and stores can improve the performance.

- Size = 100

```
* Running /usr/bin/time python final.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000241
Op Time: 0.000743
Correctness: 0.76 Model: ece408
4.76user 3.76system 0:05.09elapsed 167%CPU (0avgtext+0avgdata 2790040maxresident
)k
```

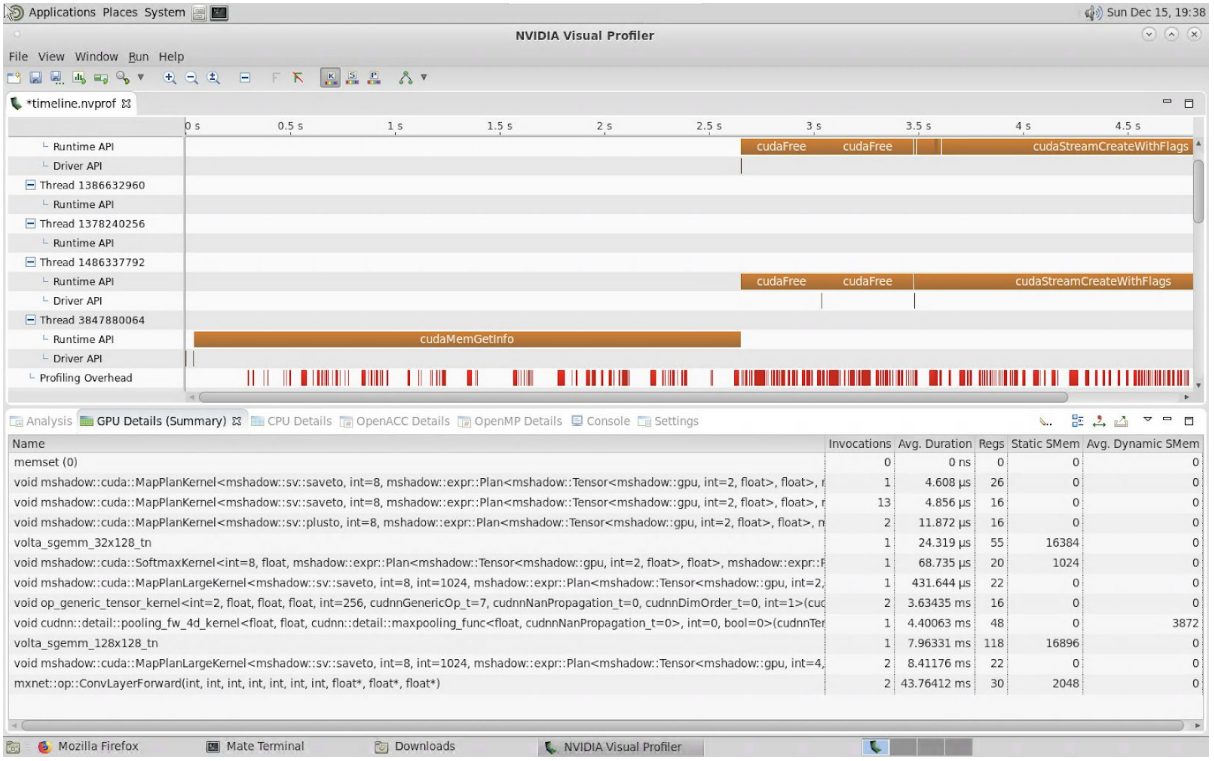
- Size = 1000:

```
* Running /usr/bin/time python final.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002141
Op Time: 0.007097
Correctness: 0.767 Model: ece408
5.01user 3.12system 0:04.61elapsed 176%CPU (0avgtext+0avgdata 2791168maxresident
)k
```

- Size = 10000:

```
* Running /usr/bin/time python final.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.021169
Op Time: 0.068800
Correctness: 0.7653 Model: ece408
5.47user 3.58system 0:05.21elapsed 173%CPU (0avgtext+0avgdata 2958336maxresident
)k
```

- Statistics and visualization on size = 10000:



Optimization 5: Input channel reduction: atomics

Although we treat the matrix in parallel, this optimization is slower due to atomic operation works as serial functions. Therefore, atomic operation is better for communication between threads in different blocks, but not good for our performance in the project.

- Size = 100 :

```
* Running /usr/bin/time python final.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.003491
Op Time: 0.004188
Correctness: 0.76 Model: ece408
8.13user 4.97system 0:09.72elapsed 134%CPU (0avgtext+0avgdata 2765892maxresident
)k
```

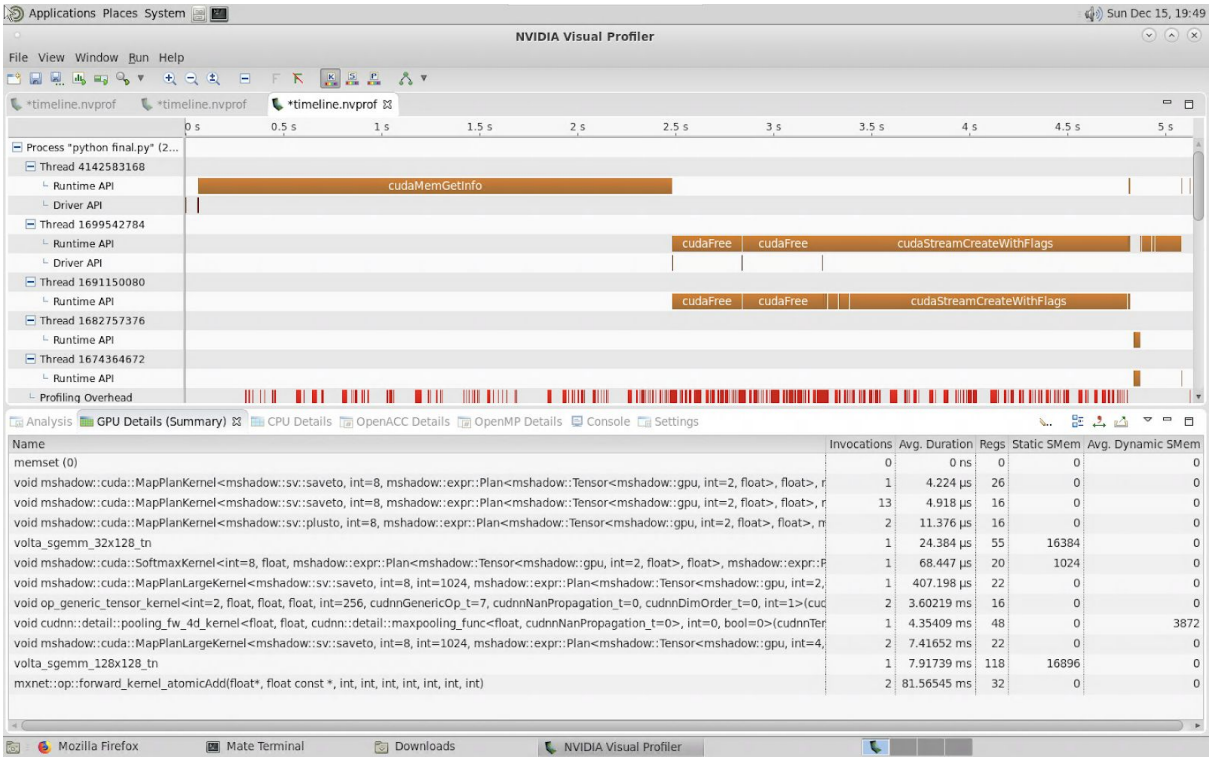
- Size = 1000:

```
* Running /usr/bin/time python final.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.003720
Op Time: 0.011611
Correctness: 0.767 Model: ece408
4.83user 3.76system 0:05.08elapsed 168%CPU (0avgtext+0avgdata 2816124maxresident
)k
```

- Size = 10000:

```
* Running /usr/bin/time python final.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.038003
Op Time: 0.126577
Correctness: 0.7653 Model: ece408
5.18user 4.20system 0:05.66elapsed 165%CPU (0avgtext+0avgdata 2975700maxresident
)k
```

- Statistics and visualization on size = 10000:



Optimization 6: Multiple kernel implementations for different layer sizes

We used a tile size of 16 in the previous tiled kernel matrix multiplication optimization, which is not the best for both the layers in our neural network. Since the number of feature maps in the first layer is 12 and that in the second layer is 24, we decided to implement two different kernels with different tile sizes. For the second layer, a tile size of 24 is perfect; while for the first layer, to take advantage of not only thread parallelism, but also memory coalescing, we chose a tile size of 16 instead of 12, so that 16 x 16 is a multiple of the warp size 32.

As we separated the two kernels and applied them to the two layers respectively, we observed a distinct reduction in the elapsed time of the second layer, which contributes to the shorter total time consumed and proves the effectiveness of this optimization.

- Size = 100：

```
* Running python final.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000248
Op Time: 0.000551
Correctness: 0.76 Model: ece408
```

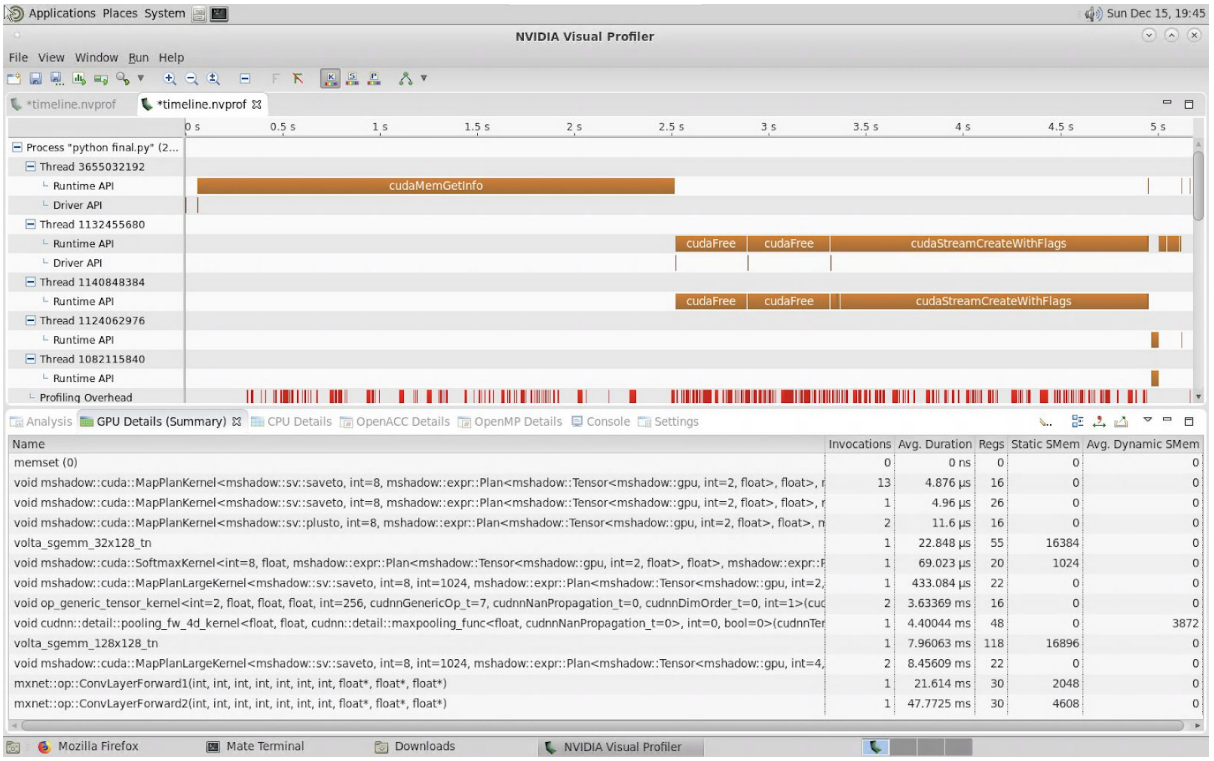
- Size = 1000:

```
* Running python final.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002144
Op Time: 0.005206
Correctness: 0.767 Model: ece408
```

- Size = 10000:

```
* Running python final.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.021170
Op Time: 0.051712
Correctness: 0.7653 Model: ece408
```

- Statistics and visualization on size = 10000:



Optimization 7: Tuning with restrict and loop unrolling (considered as one optimization only if you do both)

To further improve the kernel performances in both layers, we seek to tune the kernels in the optimization 6 with restrict and loop unrolling. Briefly, we add both const and restrict before the pointers which point to input images and weight matrices, and add restrict to the pointer which points to output feature maps. In this way, the kernel can utilize the read-only data cache on the GPUs. In addition, we add #pragma unroll before the for loops in the kernels to control loop unrolling by the GPU compiler.

- Size = 100 :

```
* Running /usr/bin/time python final.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000309
Op Time: 0.000851
Correctness: 0.76 Model: ece408
5.13user 2.76system 0:04.98elapsed 158%CPU (0avgtext+0avgdata 2802588maxresident
)k
```

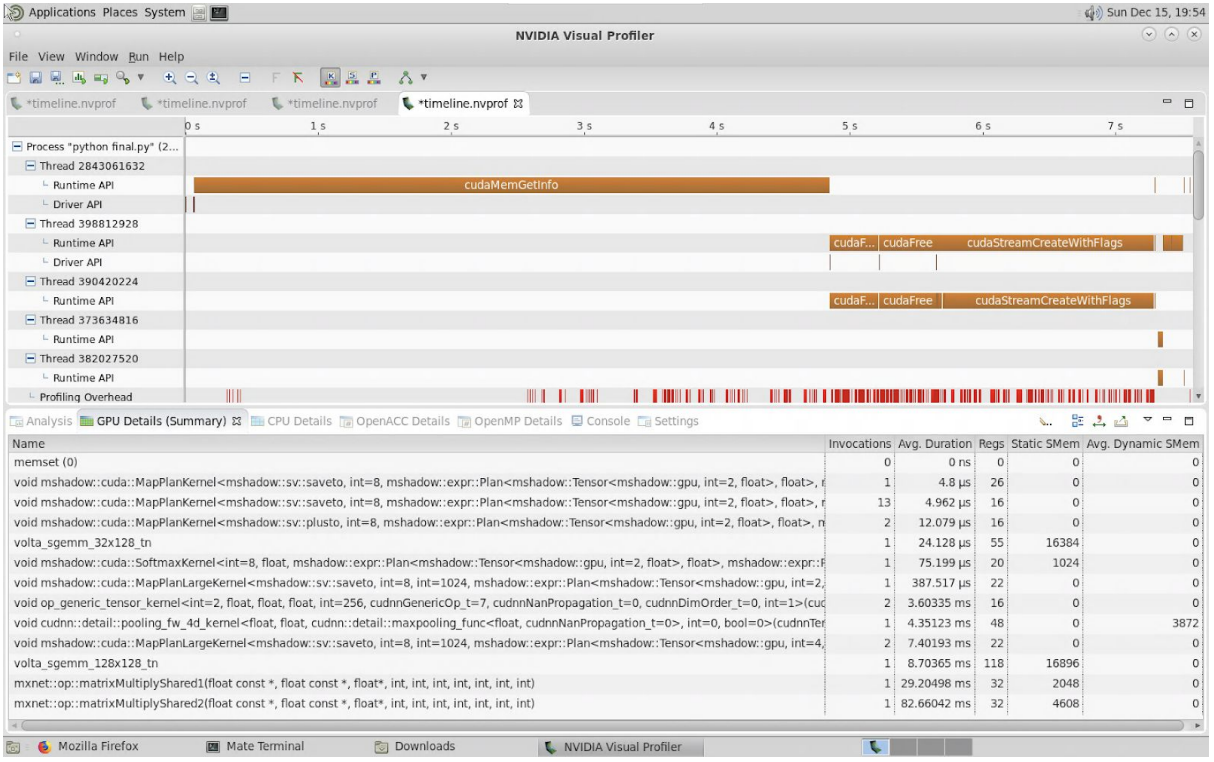
- Size = 1000:

```
* Running /usr/bin/time python final.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002625
Op Time: 0.007472
Correctness: 0.767 Model: ece408
5.20user 2.82system 0:04.85elapsed 165%CPU (0avgtext+0avgdata 2777108maxresident
)k
```

- Size = 10000:

```
* Running /usr/bin/time python final.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.028881
Op Time: 0.078382
Correctness: 0.7653 Model: ece408
5.24user 3.38system 0:05.04elapsed 171%CPU (0avgtext+0avgdata 2950608maxresident
)k
```

- Statistics and visualization on size = 10000:



This optimization does not significantly change compute and memory utilization. The kernel perfor- mances in both layers are bound by compute and memory bandwidth. Utilization of both compute and memory are balanced. The kernel is limited by the bandwidth available to the shared memory.

The enhancement of kernel performance is driven by the loop unrolling. In this way, the device can continuously execute the operations without the need to check the for loop

condition in every iteration. Utilizing the read-only data cache also improves the efficiency in data operation.

Additional optimizations

In addition to optimizing our parallel algorithm, we also did several optimizations to our final submitted code based on characteristics of the programming language and the compiler. Although these modifications are not related to the algorithm and might have a subtle effect on the performance, we made the effort just to exploit every possibility to rank higher in the final competition.

- Replace i++ with ++i: the pre-increment operator doesn't need an extra register
- Avoid modulo operator (%): replace y%x with y-(y/x)*x to avoid the time-consuming modulo operation
- Use #pragma unroll: use the macro to tell the compiler to unroll loops

Team member contributions

All members in the tensor team contributed equally in implementing code, profiling optimizations and writing the report.

ECE408 Project Milestone 4 Report

Original GPU Implement

- Size = 100:

```
==265== NVPROF is profiling process 265, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.002641
Op Time: 0.003281
Correctness: 0.76 Model: ece408
```

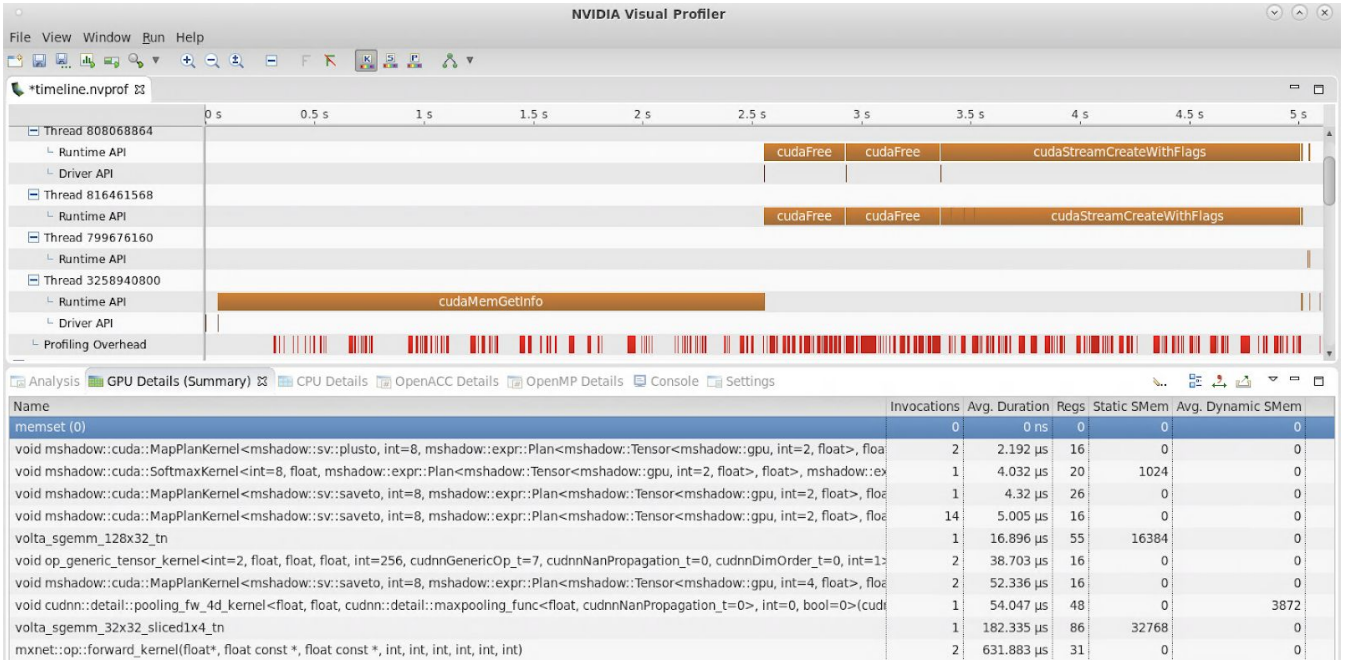
- Size = 1000:

```
==360== NVPROF is profiling process 360, command: python m3.1.py 1000
Loading model... done
New Inference
Op Time: 0.008040
Op Time: 0.022777
Correctness: 0.767 Model: ece408
```

- Size = 10000:

```
==265== NVPROF is profiling process 265, command: python m3.1.py 10000
Loading model... done
New Inference
Op Time: 0.028896
Op Time: 0.098960
Correctness: 0.7653 Model: ece408
```

- Statistics and visualization on size = 100:



Optimization 1: Shared Memory Convolution

Both Input Data X and Kernel Data W are stored into shared memory before computation. By using Shared Memory Convolution optimization, the performance of smaller size improved.

- Size = 100:

```
==268== NVPROF is profiling process 268, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000370
Op Time: 0.001172
Correctness: 0.76 Model: ece408
```

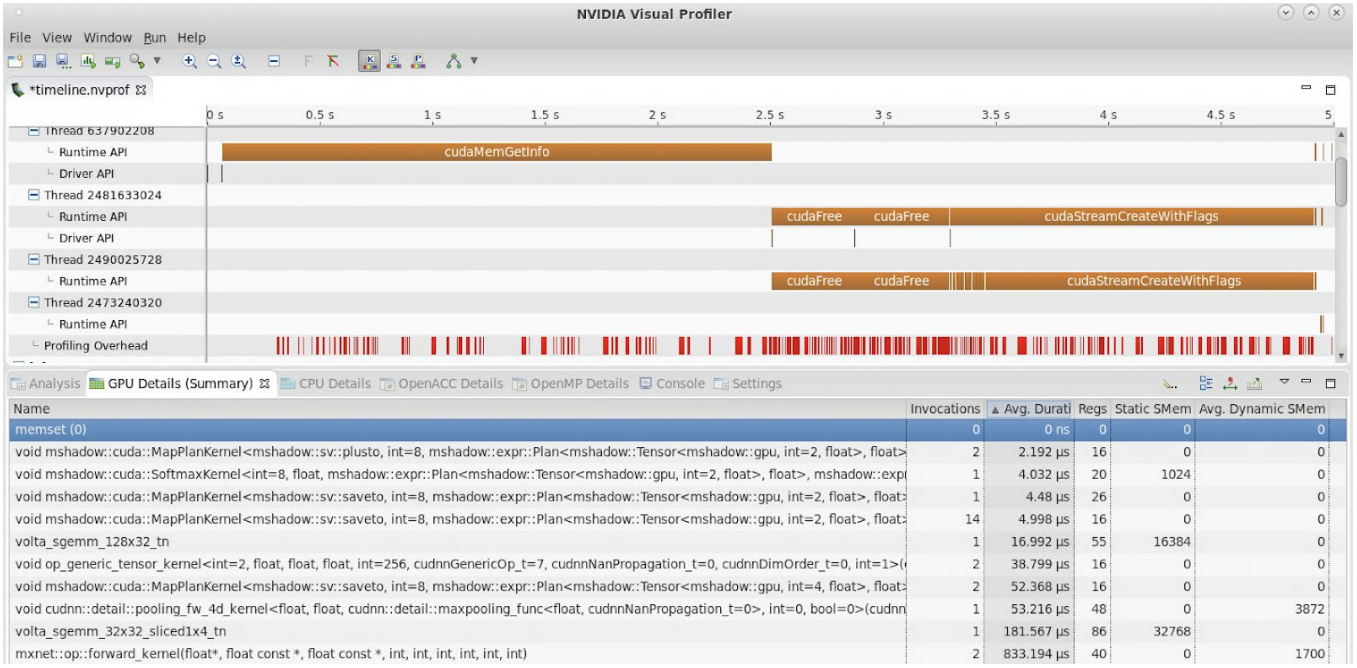
- Size = 1000:

```
==363== NVPROF is profiling process 363, command: python m3.1.py 1000
Loading model... done
New Inference
Op Time: 0.003912
Op Time: 0.011842
Correctness: 0.767 Model: ece408
```

- Size = 10000:

```
==458== NVPROF is profiling process 458, command: python m3.1.py 10000
Loading model... done
New Inference
Op Time: 0.039047
Op Time: 0.117767
Correctness: 0.7653 Model: ece408
```

- Statistics and visualization on size = 100:



Optimization 2: Shared Memory with Weight Matrix (Kernel Values) in Constant Memory

Input Data X is stored in shared memory and kernel is stored in constant memory before computation. By using constant memory optimization, the performance of smaller size improved.

- Size = 100:

```
==265== NVPROF is profiling process 265, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.002797
Op Time: 0.003546
Correctness: 0.76 Model: ece408
```

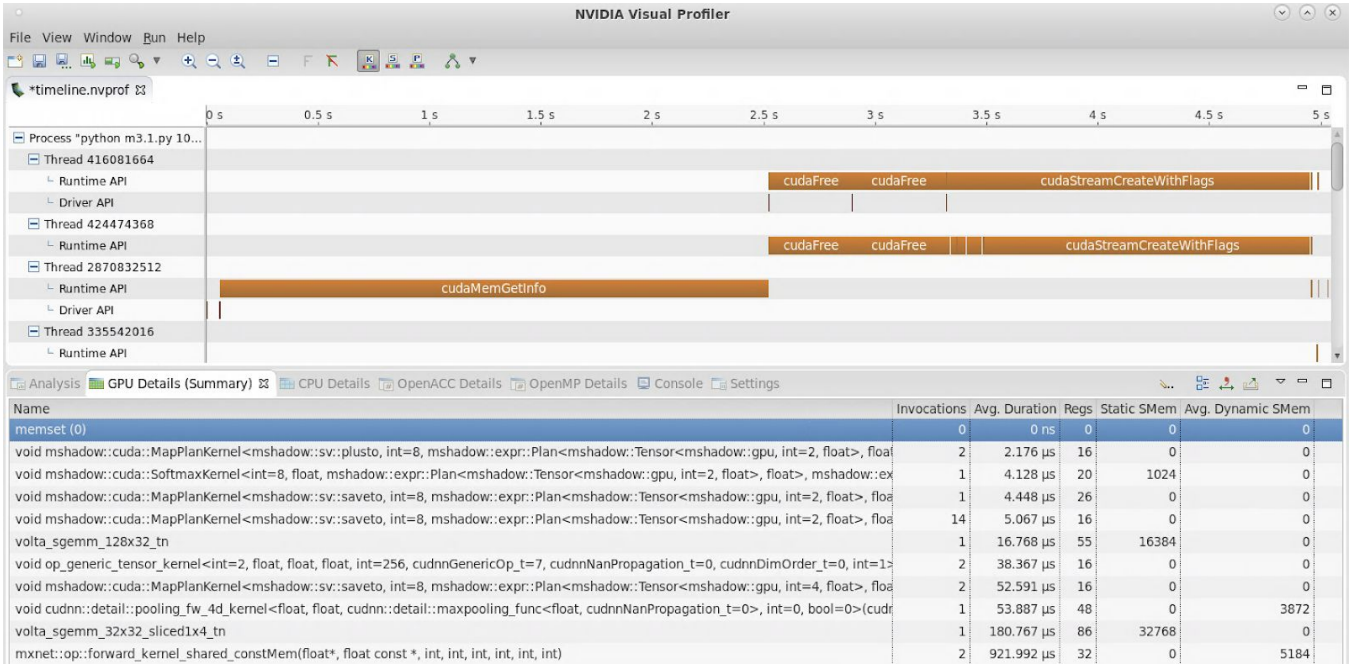
- Size = 1000

```
==360== NVPROF is profiling process 360, command: python m3.1.py 1000
Loading model... done
New Inference
Op Time: 0.008630
Op Time: 0.026650
Correctness: 0.767 Model: ece408
```

- Size = 10000

```
==455== NVPROF is profiling process 455, command: python m3.1.py 10000
Loading model... done
New Inference
Op Time: 0.086491
Op Time: 0.258496
Correctness: 0.7653 Model: ece408
```

- Statistics and visualization on size = 100:



Optimization 3: Unroll + shared-memory Matrix multiply

Both input Data X and kerneli are stored in shared memory before computation. By using unroll with shared memory, the performance is not improved. The reason from below Statistics and visualization graph is because of unroll function, more for loop and sequential commands are executed and decrease the performance.

- Size = 100:

```
* Running /usr/bin/time python m4.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002020
Op Time: 0.003060
Correctness: 0.76 Model: ece408
5.02user 2.65system 0:04.53elapsed 169%CPU (0avgtext+0avgdata 2803832maxresident
)k
```

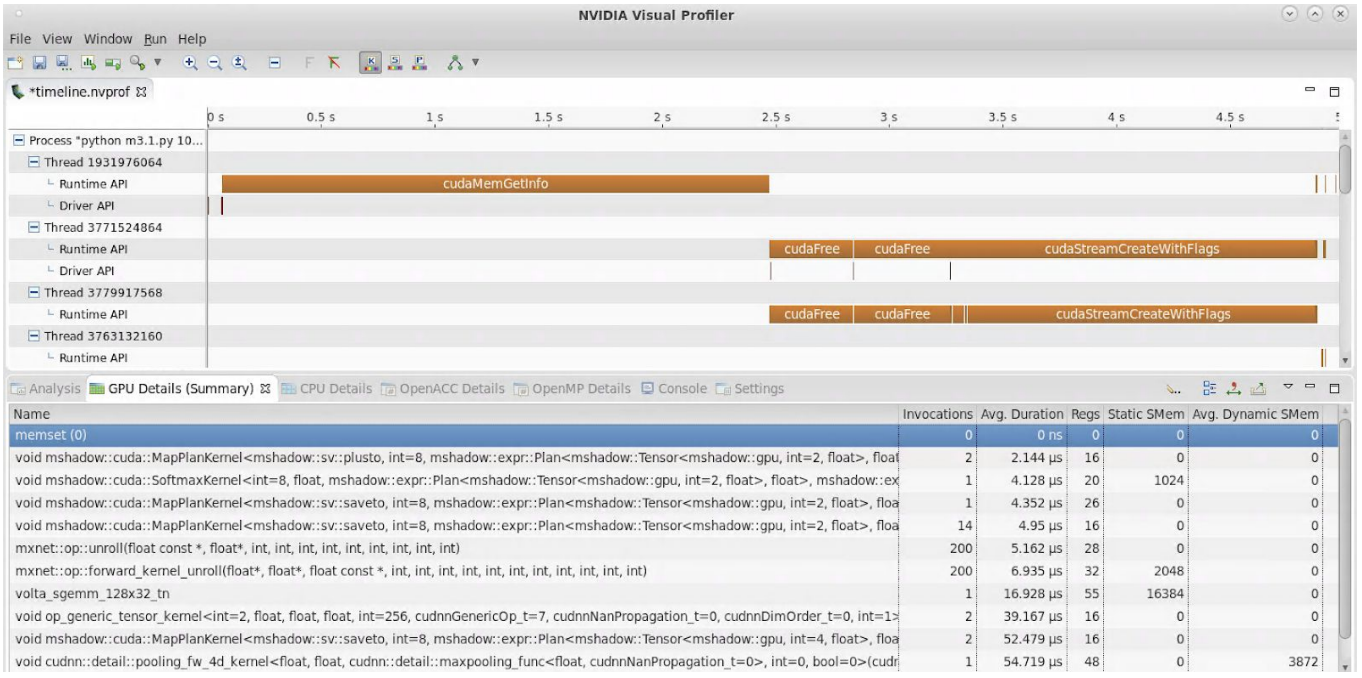
- Size = 1000

```
Op Time: 0.016756
Op Time: 0.026879
Correctness: 0.767 Model: ece408
5.22user 3.23system 0:04.97elapsed 170%CPU (0avgtext+0avgdata 2795252maxresident
)k
```

- Size = 10000

```
* Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.119580
Op Time: 0.267594
Correctness: 0.7653 Model: ece408
5.32user 3.34system 0:05.30elapsed 163%CPU (0avgtext+0avgdata 2977824maxresident
)k
```

- Statistics and visualization on size = 100:



ECE408 Project Milestone 3 Report

Shuyue Lai (shuyuel2), Yaxin Peng (yaxinp2), Jingyuan Zhang (jz61)

Team: tensor

UIUC on Campus Students

Deliverable 1: Correctness and timing with 3 different dataset sizes

✱ Running python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference

Op Time: 0.000295
Op Time: 0.000998
Correctness: 0.76 Model: ece408

✱ Running python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.003069
Op Time: 0.010254
Correctness: 0.767 Model: ece408

✱ Running python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.031567
Op Time: 0.098181
Correctness: 0.7653 Model: ece408

Deliverable 2: Demonstrate nvprof profiling the execution

GPU activities:

1. **Time(%):** 63.01% **Time:**119.68ms **Calls:**2
   Acg:59.841ms **Min:**28.956ms **Max:**90.727ms
   Name:mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
2. **Time(%):** 18.66% **Time:**35.439ms **Calls:**20
   Acg:1.7719ms **Min:**1.1200us **Max:**33.116ms
   Name:[CUDA memcpy HtoD]
3. **Time(%):** 7.80% **Time:**14.807ms **Calls:**2
   Acg:7.4035ms **Min:**2.9338ms **Max:**11.873ms
   Name: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. **Time(%):** 4.12% **Time:**7.8308ms **Calls:**1
   Avg:7.8308ms **Min:**7.8308ms **Max:**7.8308ms
   Name:volta_sgemm_128x128_tn
5. **Time(%):** 3.80% **Time:**7.2152ms **Calls:**2
   Avg:3.6076ms **Min:**24.831us **Max:**7.1904ms
   Name: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)
6. **Time(%):** 2.31% **Time:**4.3825ms **Calls:**1
   Avg:4.3825ms **Min:**4.3825ms **Max:**4.3825ms
   Name: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)
7. **Time(%):** 0.21% **Time:**391.13us **Calls:**1
   Avg:391.13us **Min:**391.13us **Max:**391.13us
   Name: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)
8. **Time(%):** 0.04% **Time:**68.000us **Calls:**1
   Avg:68.000us **Min:**68.000us **Max:**68.000us
   Name: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)
9. **Time(%):** 0.03% **Time:**65.440us **Calls:**13
   Avg:5.0330us **Min:**1.1840us **Max:**24.544us
   Name: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)
10. **Time(%):** 0.01% **Time:**26.144us **Calls:**2
    Avg:13.072us **Min:**3.8080us **Max:**22.336us
    Name: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)
11. **Time(%):** 0.01% **Time:**23.936us **Calls:**1

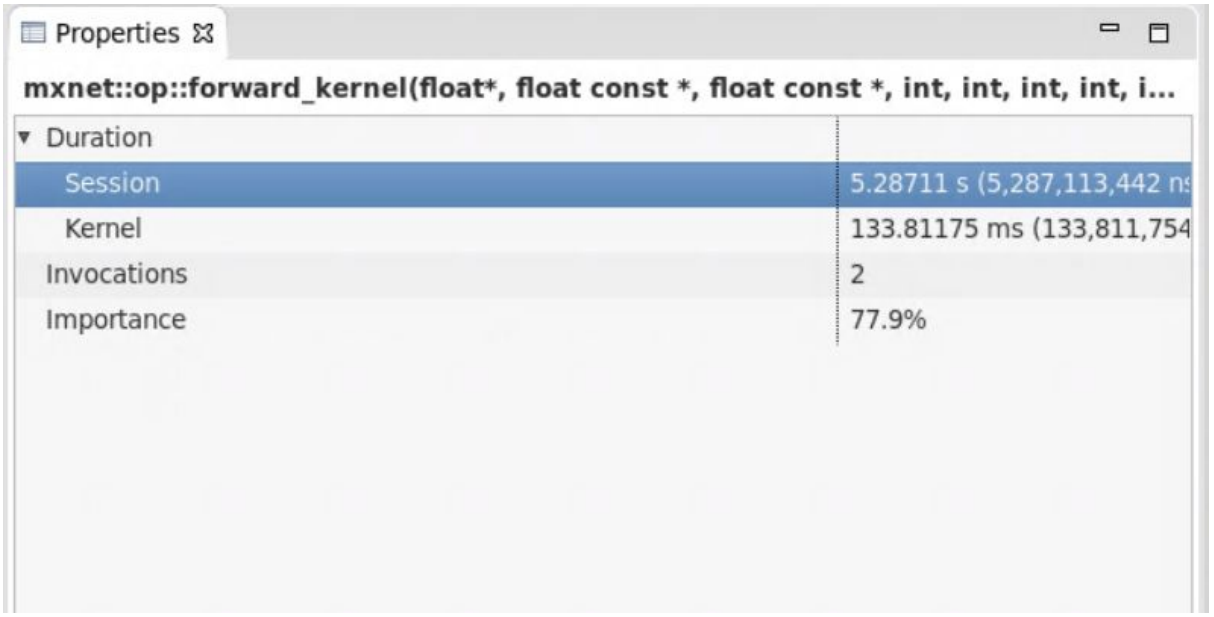
Avg:23.936us **Min:**23.936us **Max:**23.936us
Name:volta_sgemm_32x128_tn

12. **Time(%):** 0.01% **Time:**10.656us **Calls:**9
    Avg:1.1840us **Min:**992ns **Max:**1.7600us
    Name:[CUDA memset]
13. **Time(%):** 0.00% **Time:**6.8480us **Calls:**1
    Avg:6.8480us **Min:**6.8480us **Max:**6.8480us
    Name:[CUDA memcpy DtoH]
14. **Time(%):** 0.00% **Time:**4.3520us **Calls:**1
    Acg:4.3520us **Min:**4.3520us **Max:**4.3520us
    Name: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)

API calls:

```
     API calls:   41.72%  3.14080s        22  142.76ms  14.598us  1.61174s  cudaStreamCreateWithFlags
                  33.01%  2.48520s        22  112.96ms  70.384us  2.46796s  cudaMemGetInfo
20.78%  1.56442s        18  86.912ms  1.1660us  416.98ms  cudaFree
                   1.79%  134.50ms         6  22.417ms  2.6950us  90.732ms  cudaDeviceSynchronize
                   0.95%  71.254ms         9  7.9171ms  36.681us  33.159ms  cudaMemcpy2DAsync
                   0.73%  55.188ms       912  60.512us     428ns  12.503ms  cudaFuncSetAttribute
                   0.31%  23.240ms        66  352.12us  5.8380us  8.6339ms  cudaMalloc
                   0.26%  19.554ms        29  674.26us  2.3410us  10.779ms  cudaStreamSynchronize
                   0.24%  18.009ms        12  1.5008ms  7.4650us  17.568ms  cudaMemcpy
                   0.07%  5.0520ms         4  1.2630ms  431.36us  1.8761ms  cudaGetDeviceProperties
                   0.04%  2.6540ms       375  7.0770us     391ns  336.72us  cuDeviceGetAttribute
                   0.02%  1.5578ms         6  259.63us  1.6910us  1.5262ms  cudaEventCreate
                   0.02%  1.3771ms       216  6.3750us  1.1960us  161.81us  cudaEventCreateWithFlags
                   0.01%  1.0304ms         8  128.81us  14.551us  729.24us  cudaStreamCreateWithPriority
                   0.01%  760.11us         4  190.03us  90.125us  373.32us  cuDeviceTotalMem
                   0.01%  744.21us         2  372.10us  50.931us  693.28us  cudaHostAlloc
                   0.01%  698.46us         9  77.606us  10.051us  499.55us  cudaMemsetAsync
                   0.01%  647.28us         4  161.82us  95.275us  246.37us  cudaStreamCreate
                   0.01%  544.66us        27  20.172us  8.4970us  68.012us  cudaLaunchKernel
                   0.01%  385.00us       202  1.9050us     792ns  4.6310us  cudaDeviceGetAttribute
                   0.00%  293.94us         4  73.485us  47.775us  104.51us  cuDeviceGetName
                   0.00%  173.90us        29  5.9960us  1.2320us  20.578us  cudaSetDevice
                   0.00%  120.86us       557     216ns      72ns     804ns  cudaGetLastError
                   0.00%  52.222us        18  2.9010us     803ns  4.9160us  cudaGetDevice
                   0.00%  33.168us         2  16.584us  5.6730us  27.495us  cudaHostGetDevicePointer
                   0.00%  15.995us         3  5.3310us  2.8150us  7.9730us  cudaEventRecord
                   0.00%  7.2100us         6  1.2010us     573ns  2.1850us  cuDeviceGetCount
                   0.00%  7.1830us         2  3.5910us  1.8980us  5.2850us  cudaDeviceGetStreamPriorityRange
                   0.00%  5.6320us         5  1.1260us     493ns  1.8490us  cuDeviceGet
                   0.00%  5.3000us         3  1.7660us  1.0470us  3.1540us  cuInit
                   0.00%  5.2690us        20     263ns      91ns     663ns  cudaPeekAtLastError
                   0.00%  4.7080us         1  4.7080us  4.7080us  4.7080us  cuDeviceGetPCIBusId
                   0.00%  3.9050us         1  3.9050us  3.9050us  3.9050us  cudaEventQuery
                   0.00%  3.6690us         1  3.6690us  3.6690us  3.6690us  cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
                   0.00%  3.2150us         4     803ns     442ns  1.5420us  cuDeviceGetUuid
                   0.00%  2.7180us         4     679ns     257ns  1.5760us  cudaGetDeviceCount
                   0.00%  2.0550us         3     685ns     382ns  1.2700us  cuDriverGetVersion
```

NVVP for initial Performance Results：



ECE408 Project Milestone 2 Report

Shuyue Lai (shuyuel2), Yaxin Peng (yaxinp2), Jingyuan Zhang (jz61)

Team: tensor

UIUC on Campus Students

Deliverable 1: A list of all kernels that collectively consume more than 90% of the program time.

Time(%)	**Name**
30.68%	[CUDA memcpy HtoD]
17.86%	volta_scudnn_128x64_relu_interior_nn_v1
17.21%	volta_gcgemm_64x32_nt
8.77%	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)
7.82%	volta_sgemm_128x128_tn
6.58%	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)
6.50%	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%	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)

Deliverable 2: A list of all CUDA API calls that collectively consume more than 90% of the program time.

Time(%)	Name
41.36%	cudaStreamCreateWithFlags
33.22%	cudaMemGetInfo
21.25%	cudaFree
1.09%	cudaMalloc

Deliverable 3: An explanation of the difference between kernels and API calls

Cuda kernels are C functions that are executed in parallel by CUDA threads. CUDA API calls are calls made by the host code into the CUDA driver or runtime libraries, which initiate activities such as launching a kernel.

The kernel launches are asynchronous with respect to the launching thread. In other words, the Cuda API call will return immediately and the launching thread will continue to execute until it hits an explicit launch-synchronization point such as cudaDeviceSynchronize().

Deliverable 4: Output of rai running MXNet on the CPU

Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

Deliverable 5: Program run time

0:10.17

Deliverable 6: Output of rai running MXNet on the GPU

Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

Deliverable 7: Program run time

0:04.99

Create a CPU implementation

Deliverable 8: Whole program execution time

1:17.17

Deliverable 9: Op Times

First layer: 11.527422
Second layer: 61.424059