ECE 408 Final Project:

Member1: Yuxiang Li (yl48)
Member2: Tianyu Sun (tianyus2)
Member3: Chi Wang Ng (cng10)

Milestone 1:

Report: Include a list of all kernels that collectively consume more than 90% of the program time.

1. [CUDA memcpy HtoD]
2. Volta_scudnn_128x32_relu_interior_nn_v1
3. void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*,cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)
4. void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)
5. Volta_sgemm_128x128_tn
6. 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. 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. 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. 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. volta_sgemm_32x32_sliced1x4_tn

Report: Include a list of all CUDA API calls that collectively consume more than 90% of the program time.

1. cudaStreamCreateWithFlags
2. cudaMemGetInfo
3. cudaFree
4. cudaFuncSetAttribute
5. cudaMemcpy2DAsync
6. cudaStreamSynchronize
7. cudaEventCreateWithFlags
8. cudaMalloc
9. cudaGetDeviceProperties
10. cudaMemcpy

Report: Include an explanation of the difference between kernels and API calls
Kernels are executed on the device in parallel by different CUDA threads.

Kernels are built on top of a lower-level API calls, which is also accessible by the application. Kernels provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device.

Report: Show output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
19.33user 3.85system 0:13.18elapsed 175%CPU (0avgtext+0avgdata
5956088maxresident)k
0inputs+2856outputs (0major+1584783minor)pagefaults
0swaps
```

Report: List program run time
13.18s

Report: Show output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
```

```
4.49user 2.54system 0:05.01elapsed 140%CPU (0avgtext+0avgdata
2840048maxresident)k
0inputs+4568outputs (0major+704659minor)pagefaults 0swaps
```

Report: List program run time

5.01s

Milestone 2:

Data size: 10000

```
* Running /usr/bin/time python m2.1.py
Loading fashion-mnist data... done
Loading model...[22:24:03] src/nnvm/legacy_json_util.cc:204: Warning: loading symbol
00. May cause undefined behavior. Please update MXNet if you encounter any issue
done
New Inference
Op Time: 25.142652
Op Time: 151.186232
Correctness: 0.8171 Model: ece408
187.16user 7.83system 2:59.77elapsed 108%CPU (0avgtext+0avgdata 5866708maxresident)k
0inputs+0outputs (0major+2250609minor)pagefaults 0swaps
```

Milestone 3:

Data size: 100

```
New Inference
Values: 66,66,5,5,25
Op Time: 0.000477
Values: 27,27,2,2,4
Op Time: 0.001654
Correctness: 0.85 Model: ece408
4.12user 2.61system 0:04.38elapsed 153%CPU (0avgtext+0avgdata 2641584maxresident
)k
```

Data size: 1000

```
* Running /usr/bin/time python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Values: 66,66,5,5,25
Op Time: 0.004381
Values: 27,27,2,2,4
Op Time: 0.016349
Correctness: 0.827 Model: ece408
3.96user 2.33system 0:04.10elapsed 153%CPU (0avgtext+0avgdata 2644080maxresident
)k
```

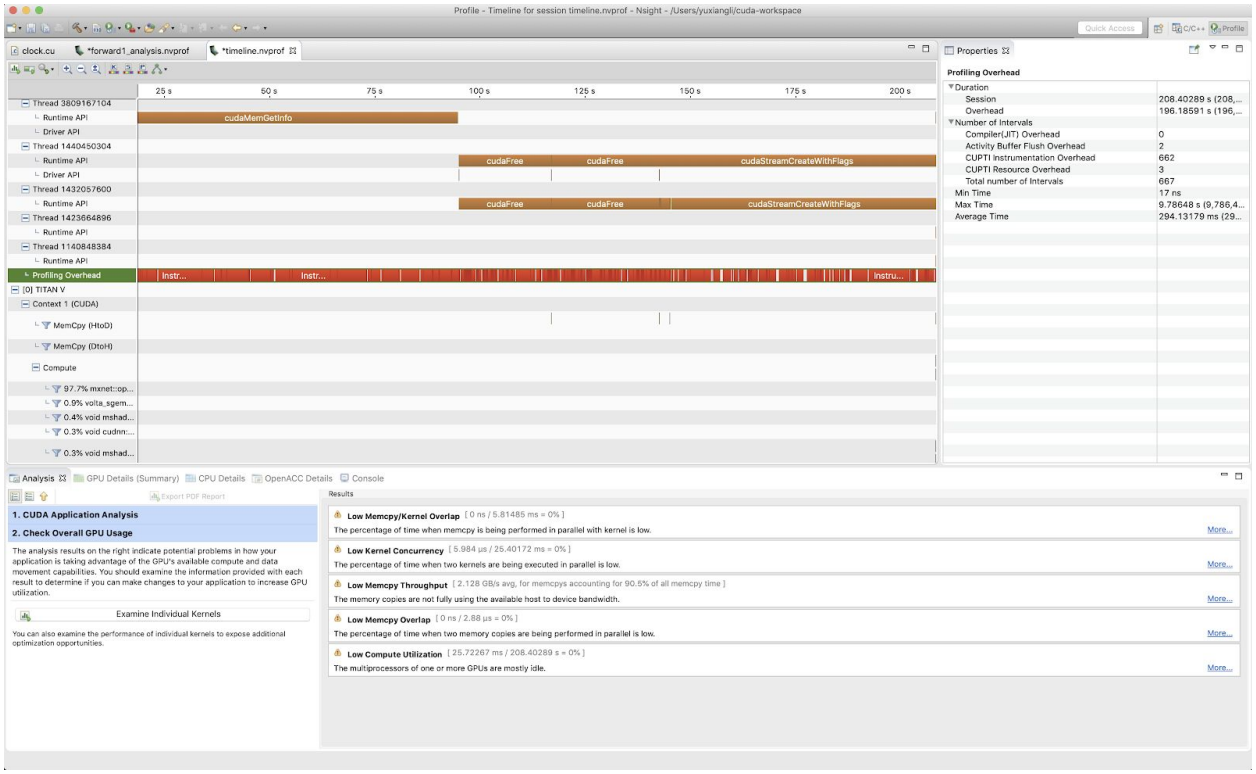
Data size : 10000

```
* Running /usr/bin/time python m3.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Values: 66,66,5,5,25
Op Time: 0.043434
Values: 27,27,2,2,4
Op Time: 0.152497
Correctness: 0.8171 Model: ece408
4.13user 2.63system 0:04.48elapsed 150%CPU (0avgtext+0avgdata 2820616maxresident
)k
```

Profile:

```
* Running nvprof python m3.1.py
Loading fashion-mnist data... done
==474== NVPROF is profiling process 474, command: python m3.1.py
Loading model... done
New Inference
Values: 66,66,5,5,25
Op Time: 0.039462
Values: 27,27,2,2,4
Op Time: 0.146472
Correctness: 0.8171 Model: ece408
==474== Profiling application: python m3.1.py
==474== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   73.24%  185.81ms         2  92.906ms  39.394ms  146.42ms  void mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu, float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int, int)
                   13.54%  34.348ms        20  1.7174ms  1.0240us  32.382ms  [CUDA memcpy HtoD]
                    5.66%  14.370ms         2  7.1850ms  2.5314ms  11.839ms  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)
                    2.88%  7.2949ms         2  3.6475ms  24.383us  7.2706ms  void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)
                    2.69%  6.8249ms         1  6.8249ms  6.8249ms  6.8249ms  volta_sgemm_128x128_tn
                    1.73%  4.3814ms         1  4.3814ms  4.3814ms  4.3814ms  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)
                    0.18%  457.79us         1  457.79us  457.79us  457.79us  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)
                    0.03%  68.384us         1  68.384us  68.384us  68.384us  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)
                    0.02%  53.471us        13  4.1130us  1.0560us  19.488us  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)
                    0.01%  30.720us         1  30.720us  30.720us  30.720us  volta_sgemm_32x32_sliced1x4_tn
                    0.01%  26.176us         2  13.088us  2.3360us  23.840us  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)
                    0.00%  7.8720us         8     984ns     928ns  1.3440us  [CUDA memset]
                    0.00%  5.5680us         1  5.5680us  5.5680us  5.5680us  [CUDA memcpy DtoH]
                    0.00%  4.6080us         1  4.6080us  4.6080us  4.6080us  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:   39.06%  2.73365s        22  124.26ms  13.371us  1.41323s  cudaStreamCreateWithFlags
                   33.95%  2.37552s        22  107.98ms  70.453us  2.37089s  cudaMemGetInfo
                   21.18%  1.48195s        18  82.331ms     319ns  389.16ms  cudaFree
                    2.86%  200.22ms         6  33.371ms  3.6600us  146.44ms  cudaDeviceSynchronize
                    1.00%  69.666ms         9  7.7406ms  28.882us  32.581ms  cudaMemcpy2DAsync
                    0.79%  55.338ms       384  144.11us     277ns  24.881ms  cudaFuncSetAttribute
                    0.57%  39.644ms        66  600.66us  5.3850us  18.116ms  cudaMalloc
                    0.26%  18.504ms        29  638.06us  4.0210us  11.033ms  cudaStreamSynchronize
                    0.17%  12.174ms       216  56.362us     321ns  10.997ms  cudaEventCreateWithFlags
                    0.07%  4.8975ms         4  1.2244ms  420.15us  1.8425ms  cudaGetDeviceProperties
                    0.03%  2.3046ms       375  6.1450us     100ns  330.44us  cuDeviceGetAttribute
                    0.01%  723.54us         2  361.77us  47.544us  676.00us  cudaHostAlloc
                    0.01%  702.04us         4  175.51us  97.339us  276.17us  cuDeviceTotalMem
                    0.01%  617.16us        27  22.857us  6.6540us  61.822us  cudaLaunchKernel
                    0.01%  521.69us        12  43.474us  5.8350us  161.53us  cudaMemcpy
                    0.01%  504.89us         4  126.22us  67.027us  218.92us  cudaStreamCreate
                    0.00%  268.88us         4  67.219us  44.463us  105.68us  cuDeviceGetName
                    0.00%  246.90us         8  30.862us  8.3700us  106.66us  cudaMemsetAsync
                    0.00%  156.17us         8  19.521us  13.190us  41.745us  cudaStreamCreateWithPriority
                    0.00%  131.99us       202     653ns     198ns  16.352us  cudaDeviceGetAttribute
                    0.00%  104.08us        29  3.5890us     515ns  11.512us  cudaSetDevice
                    0.00%  25.255us        18  1.4030us     238ns  2.9150us  cudaGetDevice
                    0.00%  22.131us         6  3.6880us     746ns  7.9230us  cudaEventCreate
                    0.00%  9.0100us         2  4.5050us  3.5650us  5.4450us  cudaEventRecord
                    0.00%  7.5560us        20     377ns     143ns     771ns  cudaPeekAtLastError
                    0.00%  4.3460us         2  2.1730us  1.6430us  2.7030us  cudaHostGetDevicePointer
                    0.00%  3.8870us         1  3.8870us  3.8870us  3.8870us  cuDeviceGetPCIBusId
                    0.00%  3.7330us         2  1.8660us  1.4170us  2.3160us  cudaDeviceGetStreamPriorityRange
                    0.00%  3.5260us         4     881ns     460ns  1.7870us  cudaGetDeviceCount
                    0.00%  2.9070us         6     484ns     145ns  1.2550us  cuDeviceGetCount
                    0.00%  2.2660us         5     453ns     206ns     876ns  cuDeviceGet
                    0.00%  2.2170us         3     739ns     470ns  1.2650us  cuInit
                    0.00%  1.7290us         3     576ns     291ns  1.1250us  cuDriverGetVersion
                    0.00%  1.4320us         5     286ns     165ns     461ns  cudaGetLastError
```

Analysis:



The performance is slow because we are reading from and writing to the global memory, which has a lower throughput. The performance is also limited because we did not reuse some of the elements and reading them from global memory again and again. For example, the filter mask never changes but we read them from the global memory every time we need them. Thus we spent more time than necessary accessing these elements and our performance is limited.

Milestone 4:

Here are the three optimizations we tried:

1. We moved the convolution mask to constant memory to speed up memory accessing
2. We accumulated the convolution sum into a local variable and only write it to global memory after we finished summing all necessary elements
3. We unrolled the two inner for loops to avoid checking loop conditions every iteration

With 1 optimization:

We first tried using only constant memory for the convolution filter to see if it improves our performance. We see that it improved performance because accessing to shared memory takes less time than global memory.

```
oading fashion-mnist data... done
==324== NVPROF is profiling process 324, command: python m4.1.py
oading model... done
New Inference
Op Time: 0.169650
Op Time: 0.710737
Correctness: 0.8171 Model: ece408
==324== Generated result file: /build/timeline.nvprof
* The build folder has been uploaded to http://s3.amazonaws.com/file
```

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

Queued	n/a
Submitted	n/a
Start	4.66064 s (4,660,635,44
End	4.83021 s (4,830,208,88
Duration	169.57344 ms (169,573,
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	26
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1

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

Queued	n/a
Submitted	n/a
Start	4.84799 s (4,847,994,90.
End	5.55869 s (5,558,689,18.
Duration	710.69428 ms (710,694,.
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	26
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 2

Low Memcpy/Kernel Overlap [0 ns / 35.70339 ms = 0%]

The percentage of time when memcpy is being performed in parallel with kernel is low.

Low Kernel Concurrency [0 ns / 887.08362 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

Low Memcpy Throughput [1.101 GB/s avg, for memcpys accounting for 0.2% of all memcpy time]

The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 7.84 µs = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

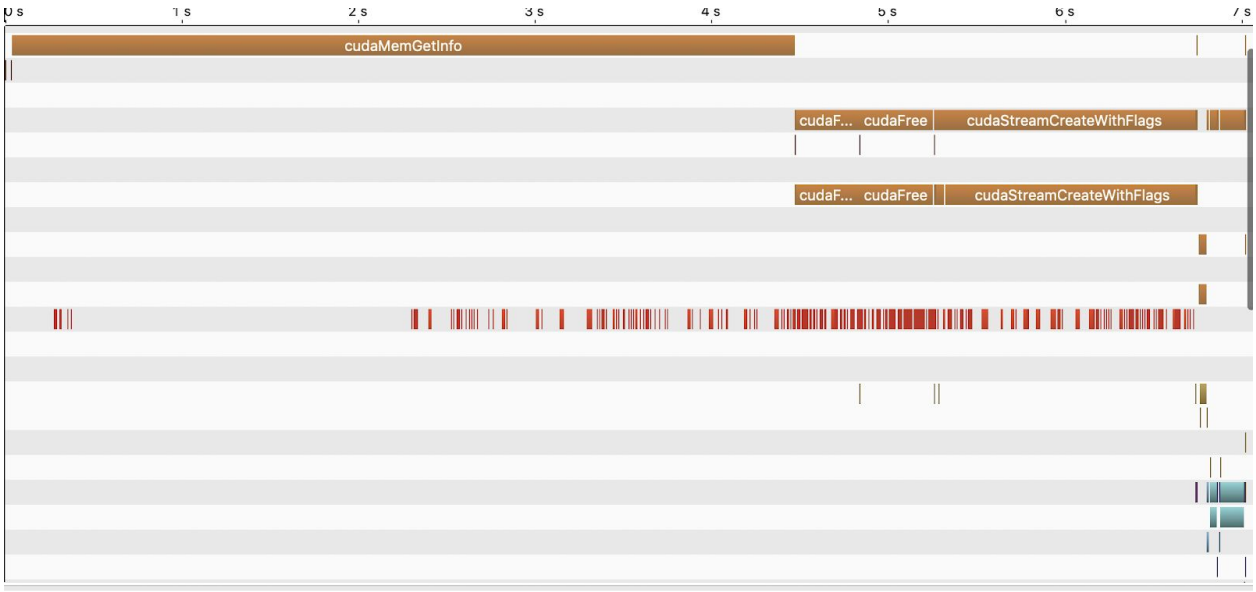
Low Compute Utilization [915.98183 ms / 5.60757 s = 16.3%]

The multiprocessors of one or more GPUs are mostly idle.

With 2 optimizations:

We then added convolution sum. We can see the Op time is decreased. This is because we reduced the amount the global memory accesses by using a local variable.

```
Loading fashion-mnist data... done
==249== NVPROF is profiling process 249, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.039346
Op Time: 0.134163
Correctness: 0.8171 Model: ece408
==249== Generated result file: /build/timeline.nvvp
```



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

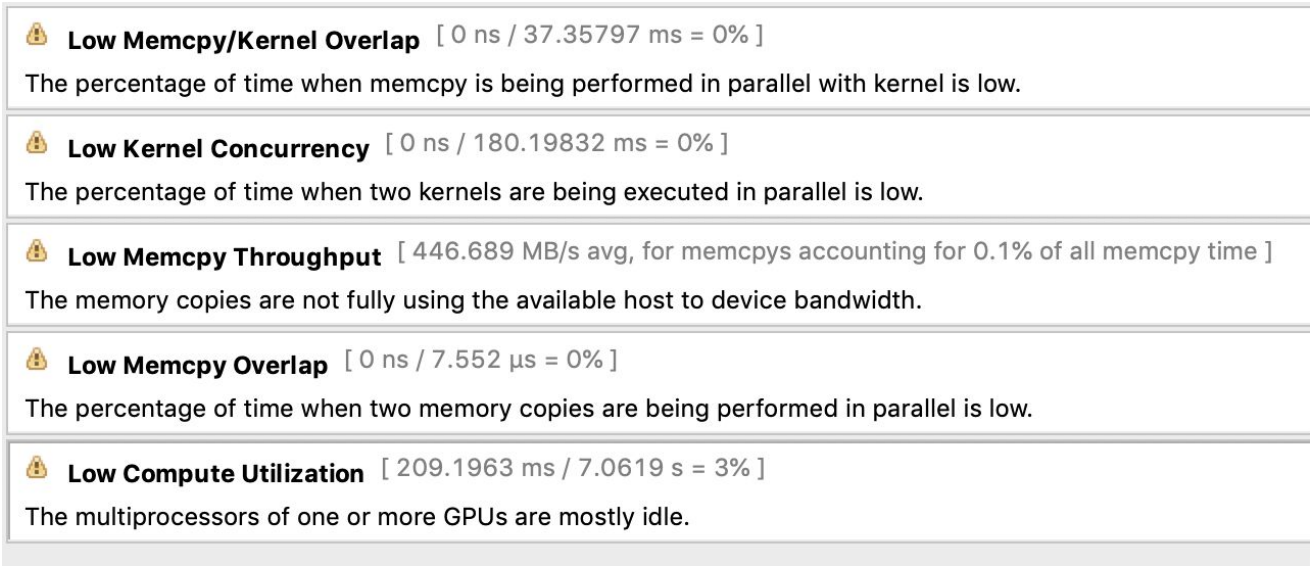
Queued	n/a
Submitted	n/a
Start	6.81639 s (6,816,391,
End	6.85565 s (6,855,654,
Duration	39.26291 ms (39,262,
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼ Occupancy	
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1

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

Queued	n/a
Submitted	n/a
Start	6.87541 s (6,875,4
End	7.00953 s (7,009,5
Duration	134.1204 ms (134,
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼ Occupancy	
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 2

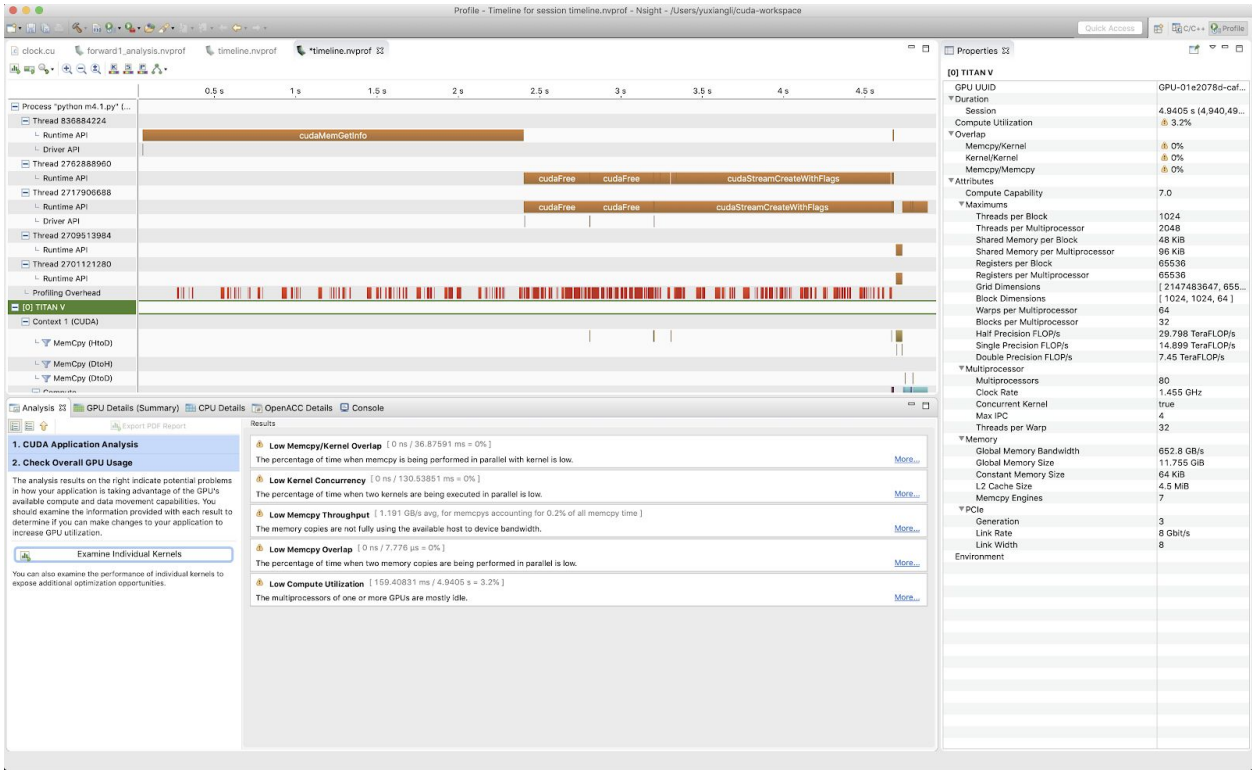


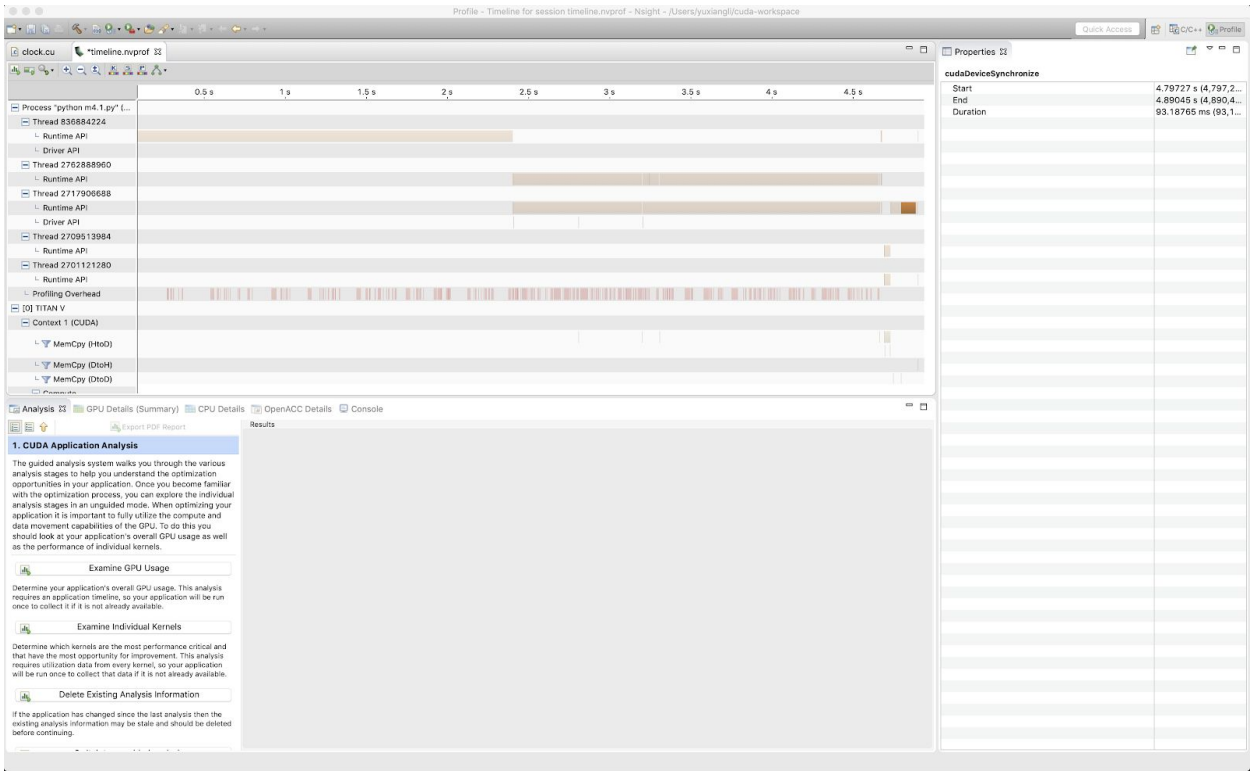
Memory and Processor Statistics

With 3 optimizations:

We then unrolled the loops in the kernel as our third optimization. The result is as follows. We can see the Op time is again significantly reduced.
Overall, the amount of overhead with optimization is significantly less than previous time (without optimization). The performance is about 30% faster with optimization.

```
* Running /usr/bin/time python m4.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.030745
Op Time: 0.098021
Correctness: 0.8171 Model: ece408
4.31user 2.57system 0:04.64elapsed 148%CPU (0avgtext+0avgdata 2826976maxresident)k
0inputs+4720outputs (
0major+702960minor)pagefaults 0swaps
```


Profile - Timeline for session timeline.nvprof - Nsight - /Users/yuxiangli/cuda-workspace
clock.cu
forward1_analysis.nvprof
timeline.nvprof
*timeline.nvprof
Properties
[0] TITAN V
GPU UUID
GPU-01e2078d-caf...
Duration
Session
4.9405 s (4,940,49...
Compute Utilization
3.2%
Overlap
Memcpy/Kernel
0%
Kernel/Kernel
0%
Memcpy/Memcpy
0%
Attributes
Compute Capability
7.0
Maximums
Threads per Block
1024
Threads per Multiprocessor
2048
Shared Memory per Block
48 KiB
Shared Memory per Multiprocessor
96 KiB
Registers per Block
65536
Registers per Multiprocessor
65536
Grid Dimensions
[2147483647, 655...
Block Dimensions
[1024, 1024, 64]
Warps per Multiprocessor
64
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
Multiprocessor
Multiprocessors
80
Clock Rate
1.455 GHz
Concurrent Kernel
true
Max IPC
4
Threads per Warp
32
Memory
Global Memory Bandwidth
652.8 GB/s
Global Memory Size
11.755 GiB
Constant Memory Size
64 KiB
L2 Cache Size
4.5 MiB
Memcpy Engines
7
PCIe
Generation
3
Link Rate
8 Gbit/s
Link Width
8
Environment
0.5 s
1 s
1.5 s
2 s
2.5 s
3 s
3.5 s
4 s
4.5 s
Process "python m4.1.py" (...
Thread 836884224
Runtime API
Driver API
Thread 2762888960
Thread 2717906688
Thread 2709513984
Thread 2701121280
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
cudaMemGetInfo
cudaFree
cudaStreamCreateWithFlags
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Export PDF Report
Results
1. CUDA Application Analysis
2. Check Overall GPU Usage
The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.
Examine Individual Kernels
You can also examine the performance of individual kernels to expose additional optimization opportunities.
Low Memcpy/Kernel Overlap [0 ns / 36.87591 ms = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low.
More...
Low Kernel Concurrency [0 ns / 130.53851 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low.
More...
Low Memcpy Throughput [1.191 GB/s avg, for memcpys accounting for 0.2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth.
More...
Low Memcpy Overlap [0 ns / 7.776 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low.
More...
Low Compute Utilization [159.40831 ms / 4.9405 s = 3.2%]
The multiprocessors of one or more GPUs are mostly idle.
More...


Profile - Timeline for session timeline.nvprof - Nsight - /Users/yuxiangli/cuda-workspace
clock.cu
*timeline.nvprof
Properties
cudaDeviceSynchronize
Start
4.79727 s (4,797,2...
End
4.89045 s (4,890,4...
Duration
93.18765 ms (93,1...
0.5 s
1 s
1.5 s
2 s
2.5 s
3 s
3.5 s
4 s
4.5 s
Process "python m4.1.py" (...
Thread 836884224
Runtime API
Driver API
Thread 2762888960
Thread 2717906688
Thread 2709513984
Thread 2701121280
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Export PDF Report
Results
1. CUDA Application Analysis
The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.
Examine GPU Usage
Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.
Examine Individual Kernels
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.
Delete Existing Analysis Information
If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.

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

Queued	n/a
Submitted	n/a
Start	4.74899 s (4,748,9...
End	4.77953 s (4,779,5...
Duration	30.5366 ms (30,53...
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1

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

Queued	n/a
Submitted	n/a
Start	4.79727 s (4,797,2...
End	4.89045 s (4,890,4...
Duration	93.1835 ms (93,18...
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 2

By unrolling the loops, the operation is accelerated.

Low Memcpy/Kernel Overlap [0 ns / 36.87591 ms = 0%]

The percentage of time when memcpy is being performed in parallel with kernel is low.

Low Kernel Concurrency [0 ns / 130.53851 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

Low Memcpy Throughput [1.191 GB/s avg, for memcpys accounting for 0.2% of all memcpy time]

The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 7.776 µs = 0%]

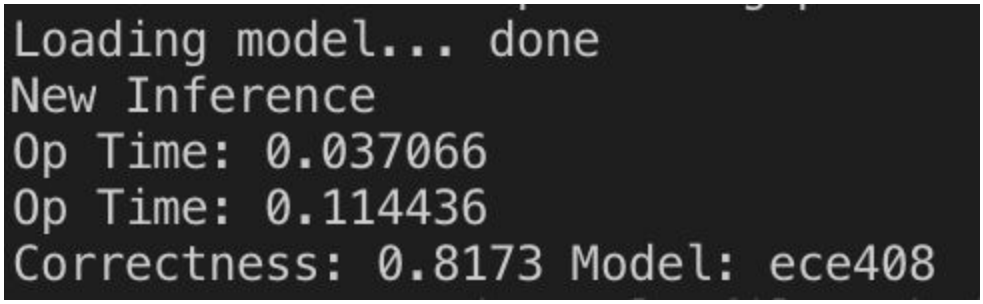
The percentage of time when two memory copies are being performed in parallel is low.

Memory and Processor Statistics

Final Report:

With 4 optimizations:

We used shared memory in this optimization, we moved the input feature matrix (x) into shared memory. However, compared with the optimization we did on last milestone, we see the performance drops (2nd Op time from 0.098021 to 0.114436). We think the reason behind this might be due to the fact that number of threads are not big enough to load, so it could be more efficient to just load the data from global memory, instead of using shared memory.



mxnet::op::forward_kernel(float*, float const *, float const *, int..	
Queued	n/a
Submitted	n/a
Start	4.66375 s (4,66...
End	4.69051 s (4,69...
Duration	26.75368 ms (2...
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

mxnet::op::forward_kernel(float*, float const *, float const *, int...	
Queued	n/a
Submitted	n/a
Start	4.70764 s (4,70...
End	4.8066 s (4,806...
Duration	98.96202 ms (9...
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1 Inference 2

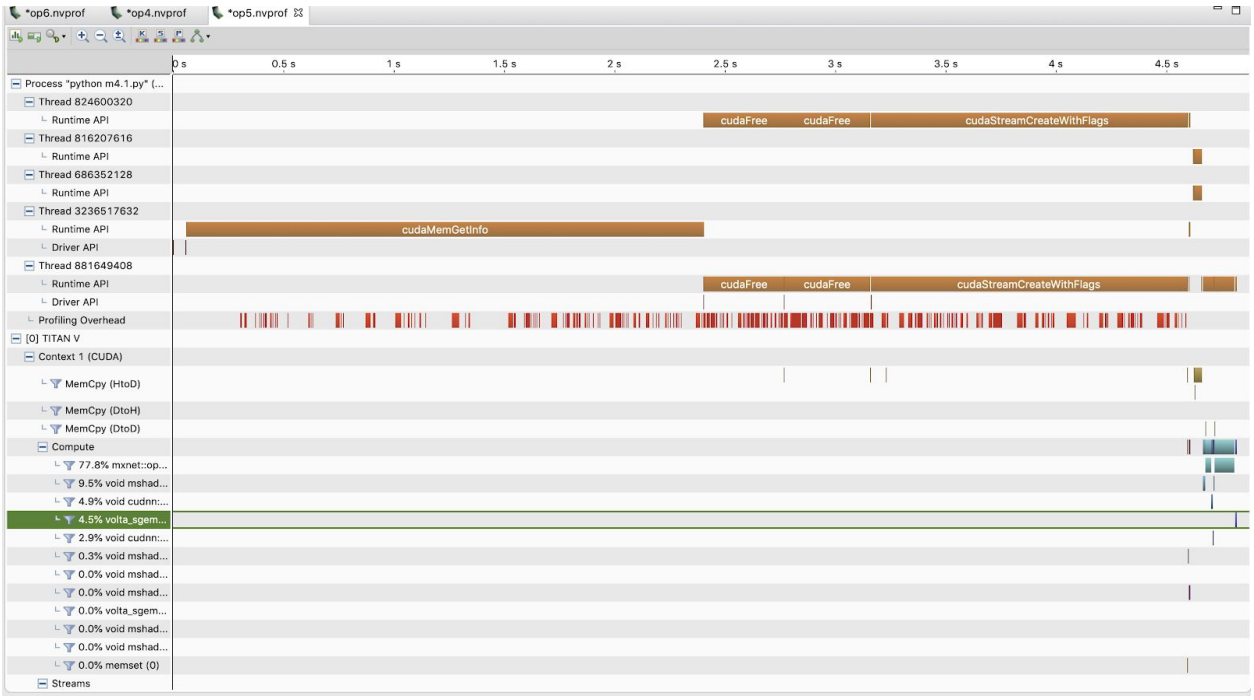
With 5 optimizations:

In this optimization, we swap the looping order while loading data from global memory to shared memory. We now load the data in column-major order from row-major order, which enable memory coalesced. Therefore, we can see the operation time is faster now.

```
New Inference
Op Time: 0.035542
Op Time: 0.099600
Correctness: 0.8168 Model: ece408
4.24user 2.65system 4.62elapsed
```

Without swap loop order:

```
done
New Inference
Op Time: 0.040549
Op Time: 0.108691
```



mxnet::op::forward_kernel(float*, float const *, float const *, int...	
Queued	n/a
Submitted	n/a
Start	4.67197 s (4,67...
End	4.69716 s (4,69...
Duration	25.1927 ms (25,...
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1

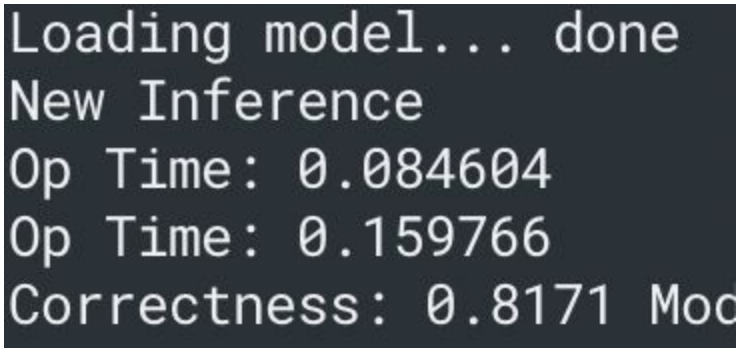
mxnet::op::forward_kernel(float*, float const *, float const *, int...	
Queued	n/a
Submitted	n/a
Start	4.71457 s (4,71...
End	4.80621 s (4,80...
Duration	91.63658 ms (9...
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 2

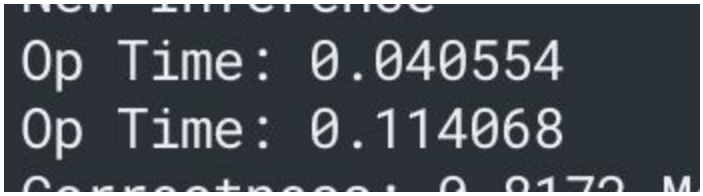
With 6 optimizations:

For the last optimization, we decided to test different parameters of tile size to see which one is performs best.

TILE_WIDTH = 32



TILE_WIDTH = 27



TILE_WIDTH = 16

```
New Inference
Op Time: 0.025252
Op Time: 0.091700
Correctness: 0.8171 Model: ece408
```

We can see the kernel performs best when TILE_WIDTH is 32. We think the reason behind this is the larger the tile size is, the utilization of shared memory is greater. Therefore, we see the performance improvement.

mxnet::op::forward_kernel(float*, float const *, float const *, int...	
Queued	n/a
Submitted	n/a
Start	4.67572 s (4,67...
End	4.71095 s (4,71...
Duration	35.23615 ms (3...
Stream	Default
Grid Size	[10000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	1.891 KiB
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 1

mxnet::op::forward_kernel(float*, float const *, float const *, int...	
Queued	n/a
Submitted	n/a
Start	4.72857 s (4,72...
End	4.82727 s (4,82...
Duration	98.697 ms (98,6...
Stream	Default
Grid Size	[10000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	1.891 KiB
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Inference 2