ECE 408 Final Project Report

Team: ParallelCorn

Xiaoming Zhao (NetID: xz23)
Chieh Hsu (NetID: chielhh2)
Bohan Zhang (NetID: bohanz2)

May 2018

1 Milestone 1

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

1. **34.05%** (118.44ms), 9 calls, void fermiPlusCgemmLDS128_batched<bool=0, bool=1, bool=0, bool=0, int=4, int=4, int=4, int=3, int=3, bool=1, bool=1>(float2**, float2**, float2**, float2*, float2 const *, float2 const *, int, int, int, int, int, int, __int64, __int64, __int64, float2 const *, float2 const *, float2, float2, int)
2. **26.98%** (93.871ms), 1 call, void cudnn::detail::implicit_convolve_sgemm<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, cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, float const *, kernel_conv_params, int, float, float, int, float const *, float const *, int, int)
3. **12.68%** (44.126ms), 9 calls, void fft2d_c2r_32x32<float, 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*)
4. **8.19%** (28.494ms), 1 call, sgemm_sm35_ldg_tn_128x8x256x16x32
5. **6.50%** (22.602ms), 14 calls, [CUDA memcpy HtoD]
6. **4.07%** (14.159ms), 2 calls, 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*)

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

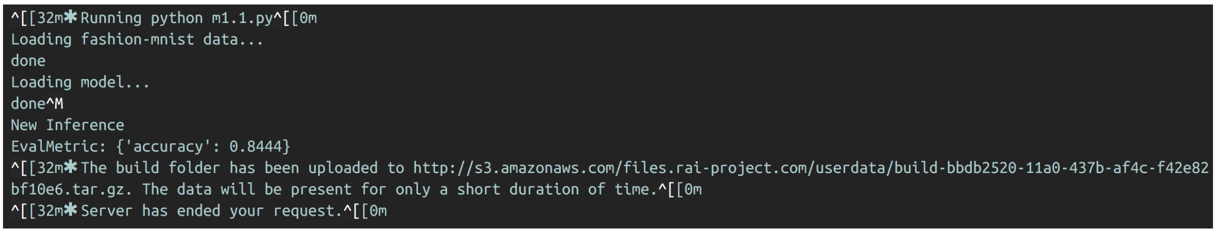
Time(%)	Time	Calls	Name
43.62%	1.94235s	18	cudaStreamCreateWithFlags
27.21%	1.21127s	10	cudaFree
20.60%	917.27ms	27	cudaMemGetInfo

Table 1: CUDA API Calls

1.3 Include an explanation of the difference between kernels and API calls

Kernels are user-coded functions that are called by the host and executed on the device (GPU, typically), whereas API calls are invoking the functions that are provided by Cuda as interface.

1.4 Show output of rai running MXNet on the CPU



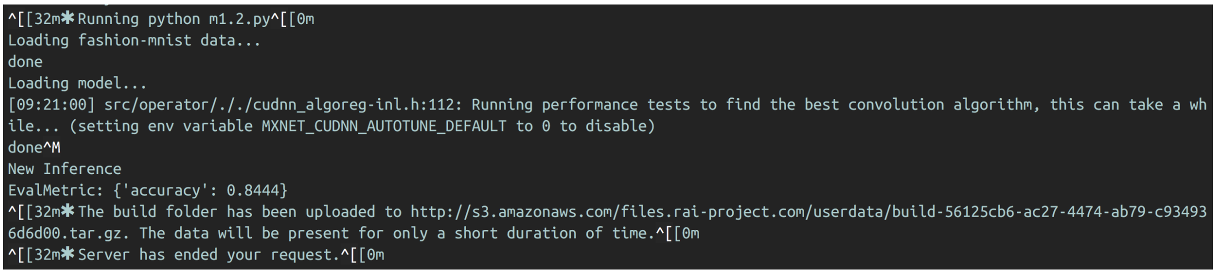
```
^[[32m✱Running python m1.1.py^[[0m
Loading fashion-mnist data...
done
Loading model...
done^M
New Inference
EvalMetric: {'accuracy': 0.8444}
^[[32m✱The build folder has been uploaded to http://s3.amazonaws.com/files.rai-project.com/userdata/build-bbdb2520-11a0-437b-af4c-f42e82
bf10e6.tar.gz. The data will be present for only a short duration of time.^[[0m
^[[32m✱Server has ended your request.^[[0m
```

Figure 1: MXNet CPU

1.5 List program run time

User: 12.67s; System: 6.27s

1.6 Show output of rai running MXNet on the GPU



```
^[[32m✱Running python m1.2.py^[[0m
Loading fashion-mnist data...
done
Loading model...
[09:21:00] src/operator/././cudnn_algoreg-inl.h:112: Running performance tests to find the best convolution algorithm, this can take a wh
ile... (setting env variable MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
done^M
New Inference
EvalMetric: {'accuracy': 0.8444}
^[[32m✱The build folder has been uploaded to http://s3.amazonaws.com/files.rai-project.com/userdata/build-56125cb6-ac27-4474-ab79-c93493
6d6d00.tar.gz. The data will be present for only a short duration of time.^[[0m
^[[32m✱Server has ended your request.^[[0m
```

Figure 2: MXNet GPU

1.7 List program run time

User: 2.30s; system: 1.10s

2 Milestone 2

2.1 Whole Program Execution Time

User: 30.48s; System: 1.48s

2.2 Op Times

First Layer Op Time: 6.570814s; Second Layer Op Time: 19.473800s

3 Milestone 3

3.1 nvprof Timeline API Calls

Time(%)	Time	Calls	Avg	Min	Max	Name
36.93%	1.93394s	18	107.44ms	23.882us	966.80ms	cudaStreamCreateWithFlags
22.91%	1.19950s	10	119.95ms	1.0020us	339.73ms	cudaFree
20.03%	1.04880s	6	174.80ms	13.403us	671.17ms	cudaDeviceSynchronize
17.80%	931.98ms	27	34.518ms	249.75us	923.94ms	cudaMemGetInfo
1.20%	62.583ms	29	2.1580ms	5.8340us	32.221ms	cudaStreamSynchronize
0.91%	47.487ms	9	5.2764ms	17.350us	22.964ms	cudaMemcpy2DAsync
0.13%	6.8965ms	45	153.26us	9.2670us	899.76us	cudaMalloc
0.03%	1.3578ms	4	339.46us	335.44us	348.66us	cuDeviceTotalMem
0.02%	1.1504ms	114	10.091us	956ns	425.89us	cudaEventCreateWithFlags
0.02%	978.26us	352	2.7790us	510ns	70.432us	cuDeviceGetAttribute
0.01%	591.66us	28	21.130us	9.3490us	76.754us	cudaLaunch
0.01%	363.96us	6	60.660us	30.285us	130.42us	cudaMemcpy
0.01%	278.61us	4	69.651us	55.444us	101.45us	cudaStreamCreate
0.00%	112.65us	168	670ns	527ns	1.6580us	cudaSetupArgument
0.00%	112.24us	104	1.0790us	854ns	1.9860us	cudaDeviceGetAttribute
0.00%	100.32us	4	25.080us	18.442us	29.777us	cuDeviceGetName
0.00%	88.815us	34	2.6120us	888ns	7.4090us	cudaSetDevice
0.00%	50.697us	2	25.348us	24.627us	26.070us	cudaStreamCreateWithPriority
0.00%	38.625us	28	1.3790us	691ns	2.4110us	cudaConfigureCall
0.00%	26.677us	10	2.6670us	1.4880us	8.6180us	cudaGetDevice
0.00%	14.908us	20	745ns	592ns	1.0340us	cudaPeekAtLastError
0.00%	6.4370us	6	1.0720us	546ns	2.4080us	cuDeviceGetCount
0.00%	5.8180us	2	2.9090us	2.8400us	2.9780us	cudaStreamWaitEvent
0.00%	5.2330us	6	872ns	635ns	1.2940us	cuDeviceGet
0.00%	5.2240us	2	2.6120us	2.5310us	2.6930us	cudaEventRecord
0.00%	4.7060us	2	2.3530us	2.0230us	2.6830us	cudaDeviceGetStreamPriorityRange
0.00%	4.4890us	5	897ns	654ns	1.1180us	cudaGetLastError
0.00%	3.4770us	3	1.1590us	1.0330us	1.2480us	cuInit
0.00%	3.4240us	1	3.4240us	3.4240us	3.4240us	cudaStreamGetPriority
0.00%	2.9860us	3	995ns	962ns	1.0470us	cuDriverGetVersion
0.00%	1.4480us	1	1.4480us	1.4480us	1.4480us	cudaGetDeviceCount

Table 2: CUDA API Calls

3.2 Top 3 Representative Profiling Result

Time(%)	Time	Calls	Avg	Min	Max	Name
90.42%	1.02679s	2	513.39ms	355.65ms	671.14ms	mxnet::op::forward_kernel
2.54%	28.823ms	1	28.823ms	28.823ms	28.823ms	sgemm_sm35_ldg_tn_128x8x256x16x32
2.08%	23.661ms	14	1.6901ms	1.5360us	22.812ms	[CUDA memcpy HtoD]

Table 3: Partial Profiling Result

3.3 Speedup with GPU

According to nvprof, the GPU convolution has the significant overall speedup when compared with the CPU implementation (0.355 on GPU vs 6.599 on CPU).

3.4 Individual Optimization

Inside the convolution kernel, the GPU code uses 16*16 tiles which enables every warp to access two consecutive memory sections, each consisting of 16 locations. This optimization utilizes 50 percent of the memory burst. On the other hand, given the relatively small block size, the kernel did not use shared memory. Thus the overhead introduced by barrier synchronization and the extra loading process is minimized for this small-block-sized convolution kernel.

3.5 NVVP Performance Result

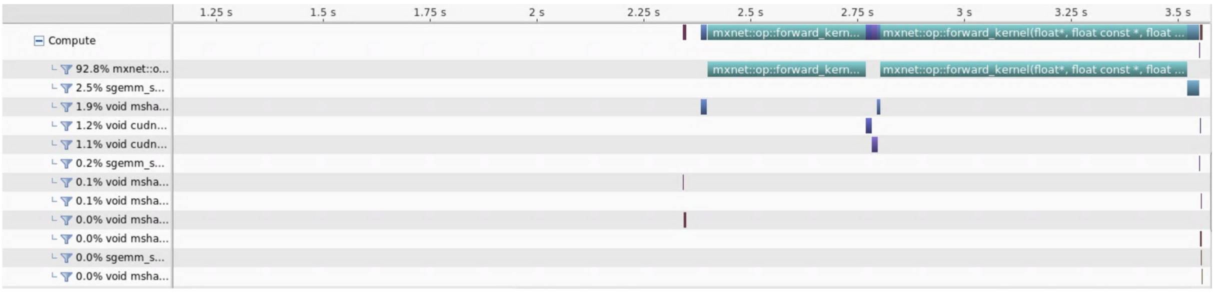


Figure 3: Kernel Performance

4 Milestone 4

4.1 Various Optimizations

Currently, we have tried the following optimizations. Pleae see table 4 for the results of gradually adding optimizations.

1. optim 1: use **constant memory** to store kernel weights
2. optim 2: use tiled **shared memory** to store input values
3. optim 3: optimize index calculation order, such as making some common part of index to the outer loop
4. optim 4: within kernel, loading data of **all input channels** into shared memory instead of using for loop over input channel
5. optim 5: write individualized and different kernels for different layers, which could reduce the calculation for generalization purposes

Layer Optimizations	layer1 (ms)	layer2 (ms)
no optim	341.328	574.938
optim 1	144.615	435.209
optim 2	150.497	505.927
optim 1 and 2	129.023	415.795
optim 1, 2 and 3	129.811	418.395
optim 1, 2, 3 and 4	82.500	204.469
optim 1, 2, 3, 4 and 5	72.938	189.648

Table 4: Speed for Optimizations

4.2 nvprof Timeline API Calls

Table 5 shows the API calls of the kernel optimizations 1, 2, 3 and 4.

Time(%)	Time	Calls	Avg	Min	Max	Name
41.78%	1.92798s	18	107.11ms	21.738us	963.67ms	cudaStreamCreateWithFlags
27.92%	1.28829s	10	128.83ms	1.3330us	378.68ms	cudaFree
21.10%	973.68ms	27	36.062ms	137.44us	969.31ms	cudaMemGetInfo
6.19%	285.44ms	6	47.573ms	13.060us	190.25ms	cudaDeviceSynchronize
1.35%	62.097ms	29	2.1413ms	5.9830us	31.610ms	cudaStreamSynchronize
1.31%	60.330ms	9	6.7034ms	13.918us	29.069ms	cudaMemcpy2DAsync
0.17%	8.0260ms	45	178.36us	10.548us	1.1864ms	cudaMalloc
0.09%	4.0975ms	4	1.0244ms	25.508us	3.9483ms	cudaStreamCreate
0.02%	1.0105ms	352	2.8700us	518ns	70.860us	cuDeviceGetAttribute
0.02%	960.21us	114	8.4220us	913ns	262.87us	cudaEventCreateWithFlags
0.02%	731.52us	4	182.88us	177.56us	194.34us	cuDeviceTotalMem
0.01%	592.70us	28	21.167us	10.879us	58.037us	cudaLaunch
0.01%	490.16us	6	81.693us	26.959us	124.88us	cudaMemcpy
0.00%	118.49us	2	59.245us	56.067us	62.424us	cudaMemcpyToSymbol
0.00%	114.86us	4	28.713us	22.836us	32.544us	cuDeviceGetName
0.00%	108.95us	154	707ns	527ns	1.8480us	cudaSetupArgument
0.00%	98.806us	104	950ns	686ns	2.1380us	cudaDeviceGetAttribute
0.00%	97.854us	34	2.8780us	931ns	21.429us	cudaSetDevice
0.00%	45.919us	2	22.959us	22.717us	23.202us	cudaStreamCreateWithPriority
0.00%	40.613us	28	1.4500us	692ns	4.3220us	cudaConfigureCall
0.00%	20.685us	10	2.0680us	1.5030us	2.6100us	cudaGetDevice
0.00%	15.768us	20	788ns	647ns	1.0930us	cudaPeekAtLastError
0.00%	6.0810us	6	1.0130us	512ns	2.0960us	cuDeviceGetCount
0.00%	5.7450us	2	2.8720us	2.4120us	3.3330us	cudaStreamWaitEvent
0.00%	4.9960us	2	2.4980us	1.5990us	3.3970us	cudaEventRecord
0.00%	4.7170us	6	786ns	636ns	981ns	cuDeviceGet
0.00%	4.3510us	3	1.4500us	1.3760us	1.4950us	cuDriverGetVersion
0.00%	4.3450us	1	4.3450us	4.3450us	4.3450us	cudaStreamGetPriority
0.00%	4.2310us	5	846ns	713ns	1.0440us	cudaGetLastError
0.00%	3.8130us	3	1.2710us	1.0530us	1.4140us	cuInit
0.00%	3.7120us	2	1.8560us	1.6660us	2.0460us	cudaDeviceGetStreamPriorityRange
0.00%	1.4690us	1	1.4690us	1.4690us	1.4690us	cudaGetDeviceCount

Table 5: CUDA API Calls

4.3 Top 4 Representative Profiling Results

Time(%)	Time	Calls	Avg	Min	Max	Name
50.38%	190.21ms	1	190.21ms	190.21ms	190.21ms	mxnet::op::forward_shareInput_constKernel30
19.39%	73.221ms	1	73.221ms	73.221ms	73.221ms	mxnet::op::forward_shareInput_constKernel64
7.82%	29.517ms	14	2.1084ms	1.5360us	28.560ms	[CUDA memcpy HtoD]
7.77%	29.339ms	1	29.339ms	29.339ms	29.339ms	sgemm_sm35_ldg_tn_128x8x256x16x32

Table 6: Partial Profiling Result

4.4 NVVP Performance Result

Please see figure 4 for NVVP evaluation.

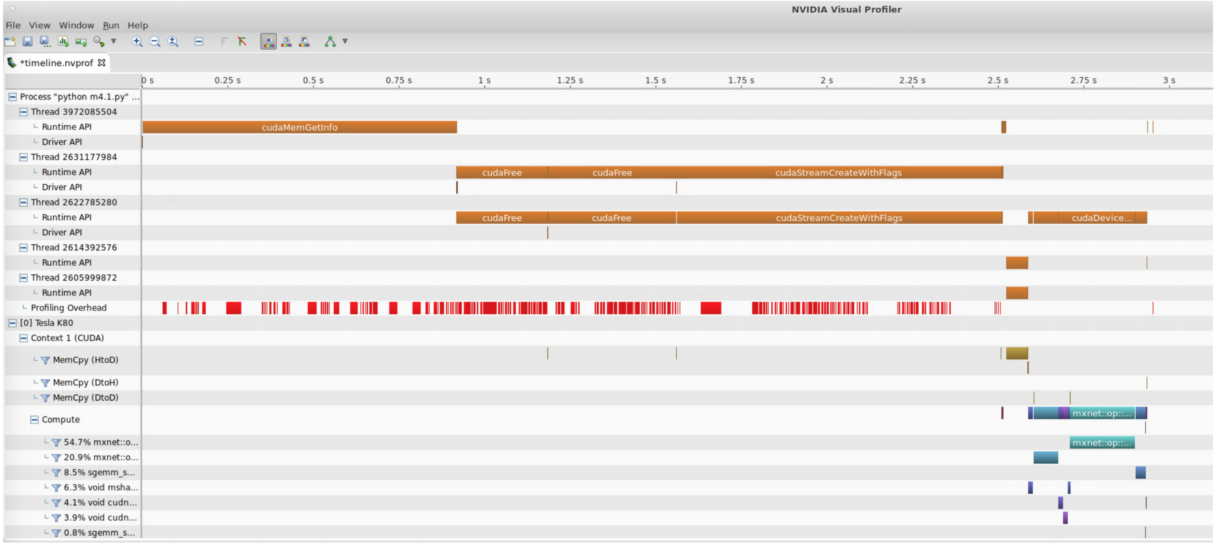


Figure 4: Kernel Performance

5 Final

5.1 Various Optimizations

This time we explore several more optimizations (optim 6 - 8) besides the ones listed in milestone 4 (optim1 - 5). Please see table 7 for speed information.

1. optim 1: use **constant memory** to store kernel weights
2. optim 2: use tiled **shared memory** to store input values
3. optim 3: optimize index calculation order, such as making some common part of index to the outer loop
4. optim 4: within kernel, loading data of **all input channels** into shared memory instead of using for loop over input channel
5. optim 5: write individualized and different kernels for **different layers**, which could reduce the calculation for generalization purposes
6. optim 6: parallelize input channel and use **atomic** operations to add results from different channels
7. optim 7: parallelize input channel and use **reduction tree** operations to sum all input channel results
8. optim 8: use **loop unroll** to reduce convolution operations to matrix multiplications

5.2 Analysis of Individual Optimizations

5.2.1 Input Channel Reduction: Atomics

Please see table 8 in appendix for nvprof information. In milestone 4, we loop over input channel within kernel. This time we want to explore whether it will be beneficial if we implement parallelism on input channel. We utilize the threads in z dimension of block to process each input channel and then sum up the values from different input channel. Because floating operations are not associative, the method we took was atomics, where each single unit non-simultaneously "update" their results.

The usage of atomic operation is a safe method to preserve the correctness; given the numerous optimizations which modify the original computation orders and the attribute of inassociatibility

Layer Optimizations	layer1 (ms)	layer2 (ms)
no optim	341.328	574.938
optim 1	144.615	435.209
optim 2	150.497	505.927
optim 1 and 2	129.023	415.795
optim 1, 2 and 3	129.811	418.395
optim 1, 2, 3 and 4	82.500	204.469
optim 1, 2, 3, 4 and 5	72.938	189.648
optim 1, 2, 3, 4 and 6	581.456	437.595
optim 1, 2, 3, 4 and 7	672.252	487.389
optim 8	550.885	1.0239 (s)
optim 1 and 8	511.350	800.731
optim 2 and 8	18.723 (s)	1.264 (s)
optim 1, 2 and 8	17.923 (s)	1.324 (s)

Table 7: Speed for Optimizations

of floats (meaning that float A + B + C is not always equal to B + C + A given the relatively huge different among these floats), atomic operation is a safe way because it preserves the order by enforcing each unit to queue up and submit their result one by one. On the other hand, since each unit has to wait until the previous one has done its work, the optimization did not turn out to be a good one. The time cost increases. From table 8, the API cudaDeviceSynchronize took up the major time, which verified our assumption that atomic waiting time slows down the whole script.

5.2.2 Input Channel Reduction: Trees

Please see table 9 in appendix for nvprof information. Here, we want to explore whether we will have wrong accuracy if we do not use atomic operations. For this optimization's kernel, we create a shared memory to store the results for each input channel. After all channel's result loaded into shared memeory, we used reduction tree to sum up intermediate result from all input channels to get the final result.

We could not guarantee the correctness before running it due to the non-associatibility of floating operations. However, from the result, it seems reduction tree works! Meanwhile, the time cost is even a little larger than atomic operations. From table 9, the API cudaDeviceSynchronize contributed most to time cost. We think it comes from many **control divergence** and useless operations during the final reduction part. Since in each reduction iteration, we need to fold the results and add the latter half into the front part, there will be huge control divergence and many operations for adding zero to zeros.

5.2.3 Unrolled Matrix Multiplication

Please see table 10 in appendix for nvprof information. In this optimization, instead of iteratively going through individual input **within kernel**, we merged the input of various indices into a big matrix, and used the basic matrix multiplication kernel to get the result. Typically, the convolution process is separated into three parts:

1. first unroll to create two matrices (input and mask)
2. perform the basic matrix multiplication on them
3. separate the output matrix in order to properly distribute the results into the correct output locations in individual output masks

However, eventually this optimization introduces a slowdown to our convolution. From table 10, we found that the major time cost came from launching cuda instead of implementations within kernel. We concluded that the reason might be the loop over batch of 10000. This brought us to the second optimization, hoping to resolve the problem within the matrix calculation process.

5.2.4 Unrolled Using Constant Memory

Please see table 11 in appendix for nvprof information. Since filters are always the same, we utilized constant memory to hold them, in order to reduce the overhead brought by frequent global memory access. In detail, initially given the relatively big size of all of the filters compared with the limited constant memory space provided by CUDA, we measured to determine if the constant memory is capable of storing the entire filter banks; since it is, we at the beginning copy the filter banks into constant memory, and thus this potential avoidance of global memory usage gave us a speedup. In detail, every filter is used by a specific pair (filter[i][j] is used by input[i] and output[j], for instance), and within such pair, this filter is used ceil(input_row / filter_size) * ceil(input_col / filter_size) times; given the constant memory access being significantly faster than global ones, we expected this optimization to give us a nontrivial speedup compared with the previous matrix multiplications.

However, the speedup is not obvious and from table 11, we also found that loop over batch costs us too much time.

5.2.5 Unroll with Tiled Matrix Multiplication

Please see table 12 and table 13 in appendix for nvprof information. Specifically, since the matrix multiplication portion of the Unrolled-Convolution kernel is the most computation consuming part, we used tiled matrix multiplication to replace the plain computation. In detail, we used 32 by 32 tiles. First we let all threads in the tile **coalescely** iteratively load the corresponding inputs into the shared memory, and then we perform the multiplication tile by tile using the existing elements in the shared memory.

We were expecting the speedup of this optimization majorly would come from the elimination of control divergence and memory reuse: typically, since each output element from matrix multiplication requires input_row * input_col elements, the whole output matrix requires output_row * output_col * input_row * input_col elements, but the whole kernel only does input_row * input_col global memory reads.

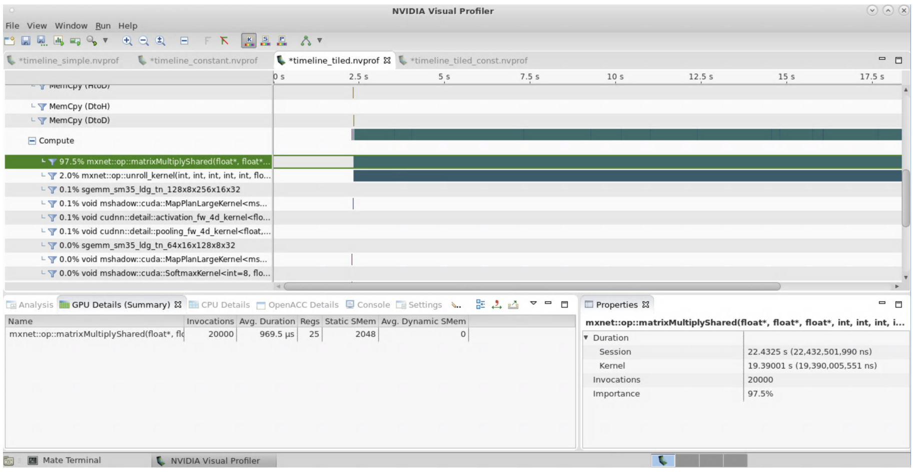
However, to our surprise, there is tremendous slowdown! From table 12 and 13, API cudaLaunch took away over 18s !!! This API is used for calling kernel. We also visualize the time cost with nvvp in figure 5. We think the reason comes from too many shared memory loading which needed to be completed. Namely, after unrolling, the input will become a huge matrix and there will be plenty of tiled blocks. For example, the first layer's unrolled matrix of each sample is 25 by 3600. With TILE_WIDTH of 16, there will be 225 blocks. For loop over all 10000 samples, there will be 2250000 **sequential** shared memory initializations! We have to give up unrolling on this test. We think that this is due to the extremely large batch size and relatively small input image size. The offset of using unrolling is too big.

6 Conclusion

We implemented 8 optimizations in this project and found that the popular **unrolling** optimizaiton performs really bad due to the large batch size and relatively small input size. This is a really important lesson that we need to use specific optimization for specific problem.

7 Appendix

7.1 nvprof Information



MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
97.5% mxnet::op::matrixMultiplyShared(float*, float*...
2.0% mxnet::op::unroll_kernel(int, int, int, int, int, flo...
0.1% sgemm_sm35_ldg_tn_128x8x256x16x32
0.1% void mshadow::cuda::MapPlanLargeKernel<ms...
0.1% void cudnn::detail::activation_fw_4d_kernel<flo...
0.1% void cudnn::detail::pooling_fw_4d_kernel<float,...
0.0% sgemm_sm35_ldg_tn_64x16x128x8x32
0.0% void mshadow::cuda::MapPlanLargeKernel<ms...
0.0% void mshadow::cuda::SoftmaxKernel<int=8, flo...

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
mxnet::op::matrixMultiplyShared(float*, fl	20000	969.5 μs	25	2048	0

mxnet::op::matrixMultiplyShared(float*, float*, float*, int, int, int, i...

▾ Duration	
Session	22.4325 s (22,432,501,990 ns)
Kernel	19.39001 s (19,390,005,551 ns)
Invocations	20000
Importance	97.5%

Figure 5: Unroll with tiled matrix multiplication

Time(%)	Time	Calls	Avg	Min	Max	Name
35.66%	1.90031s	18	105.57ms	20.029us	949.78ms	cudaStreamCreateWithFlags
23.96%	1.27698s	10	127.70ms	1.3400us	375.03ms	cudaFree
19.53%	1.04068s	6	173.45ms	15.936us	581.31ms	cudaDeviceSynchronize
18.09%	963.90ms	27	35.700ms	138.96us	959.36ms	cudaMemGetInfo
1.18%	62.941ms	29	2.1704ms	5.1200us	32.541ms	cudaStreamSynchronize
1.11%	59.063ms	9	6.5626ms	10.734us	28.481ms	cudaMemcpy2DAsync
0.24%	12.808ms	4	3.2019ms	26.417us	12.663ms	cudaStreamCreate
0.15%	8.0562ms	45	179.03us	10.135us	1.1679ms	cudaMalloc
0.02%	1.0518ms	114	9.2260us	952ns	356.34us	cudaEventCreateWithFlags
0.02%	1.0291ms	352	2.9230us	513ns	74.240us	cuDeviceGetAttribute
0.01%	744.19us	28	26.578us	10.161us	76.363us	cudaLaunch
0.01%	715.05us	4	178.76us	177.54us	181.66us	cuDeviceTotalMem
0.01%	496.14us	6	82.690us	26.585us	122.74us	cudaMemcpy
0.00%	123.89us	166	746ns	527ns	2.2260us	cudaSetupArgument
0.00%	123.26us	4	30.814us	26.414us	35.013us	cuDeviceGetName
0.00%	111.24us	2	55.620us	40.899us	70.342us	cudaMemcpyToSymbol
0.00%	106.75us	104	1.0260us	746ns	2.4860us	cudaDeviceGetAttribute
0.00%	84.682us	34	2.4900us	1.0390us	6.8380us	cudaSetDevice
0.00%	43.578us	28	1.5560us	742ns	4.0520us	cudaConfigureCall
0.00%	42.265us	2	21.132us	20.878us	21.387us	cudaStreamCreateWithPriority
0.00%	31.642us	10	3.1640us	1.6580us	7.1260us	cudaGetDevice
0.00%	17.349us	20	867ns	591ns	1.1950us	cudaPeekAtLastError
0.00%	6.0520us	2	3.0260us	2.4600us	3.5920us	cudaStreamWaitEvent
0.00%	5.8430us	6	973ns	525ns	1.7440us	cuDeviceGetCount
0.00%	5.7880us	2	2.8940us	1.7740us	4.0140us	cudaEventRecord
0.00%	5.2850us	6	880ns	621ns	1.1740us	cuDeviceGet
0.00%	4.6320us	5	926ns	671ns	1.3890us	cudaGetLastError
0.00%	4.6250us	1	4.6250us	4.6250us	4.6250us	cudaStreamGetPriority
0.00%	3.8990us	2	1.9490us	1.7390us	2.1600us	cudaDeviceGetStreamPriorityRange
0.00%	3.5910us	3	1.1970us	1.0470us	1.3760us	cuInit
0.00%	3.3260us	3	1.1080us	1.0070us	1.2590us	cuDriverGetVersion
0.00%	1.8930us	1	1.8930us	1.8930us	1.8930us	cudaGetDeviceCount

Table 8: CUDA API Calls for optim 1, 2, 3, 4 and 6

Time(%)	Time	Calls	Avg	Min	Max	Name
34.85%	1.91519s	18	106.40ms	26.682us	957.22ms	cudaStreamCreateWithFlags
23.43%	1.28774s	10	128.77ms	1.4200us	375.97ms	cudaFree
21.49%	1.18136s	6	196.89ms	17.700us	672.12ms	cudaDeviceSynchronize
17.68%	971.62ms	27	35.986ms	138.30us	967.25ms	cudaMemGetInfo
1.12%	61.789ms	29	2.1306ms	5.0460us	31.260ms	cudaStreamSynchronize
1.10%	60.511ms	9	6.7235ms	16.866us	29.081ms	cudaMemcpy2DAsync
0.15%	8.1572ms	45	181.27us	9.8600us	1.1832ms	cudaMalloc
0.09%	5.1920ms	4	1.2980ms	47.081us	4.9994ms	cudaStreamCreate
0.02%	1.1059ms	114	9.7000us	1.2940us	383.74us	cudaEventCreateWithFlags
0.02%	1.0553ms	352	2.9980us	517ns	92.669us	cuDeviceGetAttribute
0.01%	719.60us	4	179.90us	177.18us	182.17us	cuDeviceTotalMem
0.01%	610.32us	28	21.797us	10.229us	61.065us	cudaLaunch
0.01%	529.07us	6	88.178us	73.252us	105.96us	cudaMemcpy
0.00%	122.47us	4	30.616us	20.248us	38.151us	cuDeviceGetName
0.00%	121.10us	104	1.1640us	904ns	3.1390us	cudaDeviceGetAttribute
0.00%	112.82us	166	679ns	527ns	2.3100us	cudaSetupArgument
0.00%	94.417us	34	2.7760us	1.1550us	7.3260us	cudaSetDevice
0.00%	90.989us	2	45.494us	40.576us	50.413us	cudaMemcpyToSymbol
0.00%	57.066us	2	28.533us	26.637us	30.429us	cudaStreamCreateWithPriority
0.00%	43.003us	28	1.5350us	786ns	3.1990us	cudaConfigureCall
0.00%	28.794us	10	2.8790us	1.2690us	8.7890us	cudaGetDevice
0.00%	16.263us	20	813ns	646ns	1.0520us	cudaPeekAtLastError
0.00%	6.4200us	2	3.2100us	2.3940us	4.0260us	cudaStreamWaitEvent
0.00%	6.2960us	6	1.0490us	531ns	2.1330us	cuDeviceGetCount
0.00%	5.2100us	2	2.6050us	1.7630us	3.4470us	cudaEventRecord
0.00%	5.1020us	2	2.5510us	2.2500us	2.8520us	cudaDeviceGetStreamPriorityRange
0.00%	5.0970us	6	849ns	589ns	1.1400us	cuDeviceGet
0.00%	4.3830us	5	876ns	592ns	1.0570us	cudaGetLastError
0.00%	4.2700us	1	4.2700us	4.2700us	4.2700us	cudaStreamGetPriority
0.00%	3.9950us	3	1.3310us	1.2530us	1.4150us	cuInit
0.00%	3.1840us	3	1.0610us	981ns	1.1710us	cuDriverGetVersion
0.00%	1.7230us	1	1.7230us	1.7230us	1.7230us	cudaGetDeviceCount

Table 9: CUDA API Calls for optim 1, 2, 3, 4 and 7

Time(%)	Time	Calls	Avg	Min	Max	Name
33.07%	1.88258s	18	104.59ms	20.312us	940.96ms	cudaStreamCreateWithFlags
22.72%	1.29333s	10	129.33ms	1.3320us	379.07ms	cudaFree
19.29%	1.09798s	40026	27.431us	6.6500us	691.16us	cudaLaunch
17.37%	988.60ms	27	36.615ms	138.38us	984.04ms	cudaMemGetInfo
2.99%	170.27ms	300150	567ns	526ns	332.94us	cudaSetupArgument
1.80%	102.33ms	6	17.055ms	6.5390us	52.256ms	cudaDeviceSynchronize
1.07%	61.142ms	29	2.1083ms	5.1330us	30.866ms	cudaStreamSynchronize
1.01%	57.776ms	9	6.4195ms	11.307us	27.729ms	cudaMemcpy2DAsync
0.45%	25.375ms	40026	633ns	562ns	23.082us	cudaConfigureCall
0.15%	8.6900ms	47	184.89us	10.757us	1.1912ms	cudaMalloc
0.02%	1.0487ms	352	2.9790us	517ns	80.410us	cuDeviceGetAttribute
0.02%	1.0055ms	114	8.8200us	905ns	268.74us	cudaEventCreateWithFlags
0.01%	716.87us	4	179.22us	177.56us	180.85us	cuDeviceTotalMem
0.01%	508.31us	6	84.718us	29.035us	140.55us	cudaMemcpy
0.01%	439.50us	4	109.87us	25.819us	302.52us	cudaStreamCreate
0.00%	120.66us	4	30.165us	22.811us	35.687us	cuDeviceGetName
0.00%	106.82us	2	53.407us	41.940us	64.875us	cudaMemcpyToSymbol
0.00%	101.90us	104	979ns	686ns	2.3880us	cudaDeviceGetAttribute
0.00%	89.263us	34	2.6250us	880ns	7.1250us	cudaSetDevice
0.00%	43.946us	2	21.973us	21.777us	22.169us	cudaStreamCreateWithPriority
0.00%	23.331us	10	2.3330us	1.5350us	6.3720us	cudaGetDevice
0.00%	17.025us	20	851ns	664ns	1.1810us	cudaPeekAtLastError
0.00%	6.5240us	6	1.0870us	519ns	2.4460us	cuDeviceGetCount
0.00%	5.5890us	2	2.7940us	2.1630us	3.4260us	cudaStreamWaitEvent
0.00%	5.0990us	2	2.5490us	1.7820us	3.3170us	cudaEventRecord
0.00%	4.8740us	6	812ns	613ns	1.2010us	cuDeviceGet
0.00%	4.2390us	5	847ns	658ns	1.0290us	cudaGetLastError
0.00%	4.2280us	1	4.2280us	4.2280us	4.2280us	cudaStreamGetPriority
0.00%	4.0680us	2	2.0340us	1.7420us	2.3260us	cudaDeviceGetStreamPriorityRange
0.00%	3.6130us	3	1.2040us	1.1810us	1.2220us	cuInit
0.00%	2.9180us	3	972ns	880ns	1.1020us	cuDriverGetVersion
0.00%	1.9210us	1	1.9210us	1.9210us	1.9210us	cudaGetDeviceCoun

Table 10: CUDA API Calls for optim 8

Time(%)	Time	Calls	Avg	Min	Max	Name
34.92%	1.80298s	18	100.17ms	19.655us	901.24ms	cudaStreamCreateWithFlags
23.13%	1.19455s	10	119.46ms	1.2480us	351.16ms	cudaFree
17.37%	896.83ms	27	33.216ms	138.30us	892.31ms	cudaMemGetInfo
16.84%	869.71ms	40026	21.728us	6.4610us	531.59us	cudaLaunch
3.17%	163.50ms	280150	583ns	423ns	605.04us	cudaSetupArgument
1.72%	88.689ms	6	14.782ms	8.3510us	40.762ms	cudaDeviceSynchronize
1.19%	61.706ms	29	2.1278ms	5.6600us	31.343ms	cudaStreamSynchronize
0.92%	47.361ms	9	5.2623ms	9.1590us	22.810ms	cudaMemcpy2DAsync
0.50%	25.777ms	40026	644ns	545ns	205.38us	cudaConfigureCall
0.16%	8.2073ms	47	174.62us	9.1260us	1.1675ms	cudaMalloc
0.02%	1.0065ms	352	2.8590us	515ns	70.672us	cuDeviceGetAttribute
0.02%	911.56us	114	7.9960us	906ns	249.16us	cudaEventCreateWithFlags
0.02%	852.40us	4	213.10us	24.214us	725.69us	cudaStreamCreate
0.01%	713.09us	4	178.27us	176.62us	181.03us	cuDeviceTotalMem
0.01%	456.03us	6	76.005us	29.214us	133.19us	cudaMemcpy
0.00%	115.59us	4	28.896us	23.991us	33.576us	cuDeviceGetName
0.00%	97.767us	104	940ns	687ns	1.9380us	cudaDeviceGetAttribute
0.00%	85.645us	34	2.5180us	855ns	7.0030us	cudaSetDevice
0.00%	78.349us	2	39.174us	34.583us	43.766us	cudaMemcpyToSymbol
0.00%	42.378us	2	21.189us	20.525us	21.853us	cudaStreamCreateWithPriority
0.00%	22.974us	10	2.2970us	1.4520us	6.4870us	cudaGetDevice
0.00%	16.520us	20	826ns	672ns	1.0510us	cudaPeekAtLastError
0.00%	5.9530us	6	992ns	555ns	2.0050us	cuDeviceGetCount
0.00%	5.6080us	2	2.8040us	2.2380us	3.3700us	cudaStreamWaitEvent
0.00%	4.5390us	6	756ns	600ns	1.0960us	cuDeviceGet
0.00%	4.5100us	2	2.2550us	1.6430us	2.8670us	cudaEventRecord
0.00%	4.3770us	5	875ns	658ns	1.1360us	cudaGetLastError
0.00%	3.7230us	1	3.7230us	3.7230us	3.7230us	cudaStreamGetPriority
0.00%	3.3990us	2	1.6990us	1.4800us	1.9190us	cudaDeviceGetStreamPriorityRange
0.00%	3.3620us	3	1.1200us	964ns	1.2040us	cuInit
0.00%	3.0570us	3	1.0190us	885ns	1.0980us	cuDriverGetVersion
0.00%	1.7620us	1	1.7620us	1.7620us	1.7620us	cudaGetDeviceCount

Table 11: CUDA API Calls for optim 1 and 8

Time(%)	Time	Calls	Avg	Min	Max	Name
77.81%	18.5570s	40026	463.62us	5.8540us	2.1869ms	cudaLaunch
7.60%	1.81282s	18	100.71ms	26.696us	906.08ms	cudaStreamCreateWithFlags
5.10%	1.21619s	10	121.62ms	1.2290us	354.76ms	cudaFree
4.37%	1.04326s	6	173.88ms	8.2120us	956.80ms	cudaDeviceSynchronize
3.75%	894.87ms	27	33.143ms	140.03us	890.36ms	cudaMemGetInfo
0.75%	179.99ms	320150	562ns	311ns	203.33us	cudaSetupArgument
0.26%	61.844ms	29	2.1325ms	5.7010us	31.507ms	cudaStreamSynchronize
0.19%	44.420ms	9	4.9355ms	11.402us	21.546ms	cudaMemcpy2DAsync
0.10%	24.572ms	40026	613ns	318ns	193.29us	cudaConfigureCall
0.04%	8.6364ms	47	183.75us	8.8410us	1.1960ms	cudaMalloc
0.00%	1.0955ms	114	9.6100us	1.1410us	317.25us	cudaEventCreateWithFlags
0.00%	1.0198ms	352	2.8970us	516ns	73.454us	cuDeviceGetAttribute
0.00%	714.86us	4	178.71us	177.40us	180.54us	cuDeviceTotalMem
0.00%	522.21us	6	87.035us	31.158us	155.38us	cudaMemcpy
0.00%	269.32us	4	67.330us	25.380us	124.97us	cudaStreamCreate
0.00%	120.81us	4	30.201us	20.156us	37.371us	cuDeviceGetName
0.00%	118.97us	104	1.1430us	860ns	3.3000us	cudaDeviceGetAttribute
0.00%	98.427us	2	49.213us	45.587us	52.840us	cudaMemcpyToSymbol
0.00%	83.476us	34	2.4550us	879ns	8.8350us	cudaSetDevice
0.00%	60.111us	2	30.055us	24.994us	35.117us	cudaStreamCreateWithPriority
0.00%	34.543us	10	3.4540us	1.1950us	9.2210us	cudaGetDevice
0.00%	14.458us	20	722ns	612ns	1.0040us	cudaPeekAtLastError
0.00%	7.1180us	6	1.1860us	571ns	2.4100us	cuDeviceGetCount
0.00%	5.4570us	1	5.4570us	5.4570us	5.4570us	cudaStreamGetPriority
0.00%	5.3120us	2	2.6560us	2.1410us	3.1710us	cudaStreamWaitEvent
0.00%	5.0710us	6	845ns	604ns	1.0530us	cuDeviceGet
0.00%	5.0350us	2	2.5170us	1.4060us	3.6290us	cudaEventRecord
0.00%	4.7260us	2	2.3630us	1.9590us	2.7670us	cudaDeviceGetStreamPriorityRange
0.00%	4.4470us	5	889ns	693ns	1.1050us	cudaGetLastError
0.00%	4.1750us	3	1.3910us	1.3650us	1.4320us	cuInit
0.00%	3.5560us	3	1.1850us	1.0440us	1.3520us	cuDriverGetVersion
0.00%	1.5330us	1	1.5330us	1.5330us	1.5330us	cudaGetDeviceCount

Table 12: CUDA API Calls for optim 2 and 8

Time(%)	Time	Calls	Avg	Min	Max	Name
76.17%	17.8620s	40026	446.26us	6.2740us	2.0139ms	cudaLaunch
8.32%	1.95146s	18	108.41ms	19.831us	974.95ms	cudaStreamCreateWithFlags
5.53%	1.29586s	10	129.59ms	1.3590us	378.34ms	cudaFree
4.29%	1.00716s	6	167.86ms	8.2480us	917.60ms	cudaDeviceSynchronize
4.25%	997.65ms	27	36.950ms	139.32us	993.08ms	cudaMemGetInfo
0.75%	176.89ms	300150	589ns	488ns	335.51us	cudaSetupArgument
0.26%	61.261ms	29	2.1124ms	4.8600us	31.012ms	cudaStreamSynchronize
0.25%	59.784ms	9	6.6426ms	9.3840us	28.762ms	cudaMemcpy2DAsync
0.11%	25.831ms	40026	645ns	548ns	14.614us	cudaConfigureCall
0.04%	9.0887ms	47	193.38us	11.485us	1.1982ms	cudaMalloc
0.00%	1.0625ms	352	3.0180us	514ns	84.669us	cuDeviceGetAttribute
0.00%	739.52us	4	184.88us	178.60us	202.10us	cuDeviceTotalMem
0.00%	726.66us	114	6.3740us	914ns	196.33us	cudaEventCreateWithFlags
0.00%	399.31us	6	66.551us	25.159us	148.63us	cudaMemcpy
0.00%	275.23us	4	68.807us	25.147us	141.18us	cudaStreamCreate
0.00%	127.13us	4	31.782us	21.899us	38.986us	cuDeviceGetName
0.00%	115.09us	2	57.546us	48.337us	66.756us	cudaMemcpyToSymbol
0.00%	103.70us	104	997ns	687ns	2.3870us	cudaDeviceGetAttribute
0.00%	92.191us	34	2.7110us	912ns	7.3170us	cudaSetDevice
0.00%	43.939us	2	21.969us	20.959us	22.980us	cudaStreamCreateWithPriority
0.00%	32.731us	10	3.2730us	1.5260us	8.0470us	cudaGetDevice
0.00%	17.211us	20	860ns	602ns	1.4910us	cudaPeekAtLastError
0.00%	6.3540us	6	1.0590us	499ns	2.3760us	cuDeviceGetCount
0.00%	5.8040us	2	2.9020us	2.3100us	3.4940us	cudaStreamWaitEvent
0.00%	5.3880us	6	898ns	616ns	1.6930us	cuDeviceGet
0.00%	5.0040us	2	2.5020us	1.5070us	3.4970us	cudaEventRecord
0.00%	4.5040us	5	900ns	596ns	1.1640us	cudaGetLastError
0.00%	4.3730us	1	4.3730us	4.3730us	4.3730us	cudaStreamGetPriority
0.00%	3.9910us	3	1.3300us	1.1670us	1.5030us	cuInit
0.00%	3.8850us	2	1.9420us	1.6130us	2.2720us	cudaDeviceGetStreamPriorityRange
0.00%	3.2140us	3	1.0710us	947ns	1.2190us	cuDriverGetVersion
0.00%	1.7770us	1	1.7770us	1.7770us	1.7770us	cudaGetDeviceCount

Table 13: CUDA API Calls for optim 1, 2 and 8