Team Profile
Name: cudafree
Member1: Chenwei Zhang NetID: chenwei6
Member2: Tinghui Liao NetID: tinghui4
Member3: Kun Qiao NetID: kunqiao2
Affiliation: UIUC

Milestone 1

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

Time (%)	Name
37.23%	CUDA memcpy HtoD
22.28%	volta_scudnn_128x32_relu_interior_nn_v1
21.16%	void cudnn::detail::implicit_convolve_sgemm<…>(…)
7.40%	void cudnn::detail::activation_fw_4d_kernel<…>(…)
6.83%	volta_sgemm_128x128_tn
4.39%	void cudnn::detail::pooling_fw_4d_kernel<…>(…)

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

Time (%)	Name
40.92%	cudaStreamCreateWithFlags
33.41%	cudaMemGetInfo
22.44%	cudaFree
1.06%	cudaMemcpy2DAsync
0.81%	cudaStreamSynchronize

3. Explanation of the difference between kernels and API calls.

Kernels are low-level computer programs interfacing with the hardware (i.e. GPU) on top of which applications are running. Kernels could also manage input/output requests from software.

But API calls are sets of protocols, subroutine definitions, and tools for building application software. API calls will provide interfaces that developers should use when writing code using libraries and a kind of programming language.

4. Show output of rai running MXNet on the CPU

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

5. List program run time(CPU)

19.88user 3.70system 0:13.53elapsed 174%CPU (0avgtext+0avgdata 5956128maxresident)k
0inputs+2856outputs (0major+1585668minor)pagefaults 0swaps

6. Show output of rai running MXNet on the GPU

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

7. List program run time(GPU)

4.34user 2.44system 0:04.75elapsed 142%CPU (0avgtext+0avgdata 2849272maxresident)k
0inputs+4568outputs (0major+706602minor)pagefaults 0swaps

Milestone 2

1. Create a CPU implementation

```
void forward(mshadow::Tensor<cpu, 4, DType> &y, const mshadow::Tensor<cpu, 4, DType> &x,
const mshadow::Tensor<cpu, 4, DType> &k)
{
   const int B = x.shape_[0];
   const int M = y.shape_[1];
   const int C = x.shape_[1];
   const int H = x.shape_[2];
   const int W = x.shape_[3];
   const int K = k.shape_[3];

   int H_out = H - K + 1;
   int W_out = W - K + 1;

   for (int b = 0; b < B; ++b)
      for(int m = 0;  m < M;  m++)
         for(int h = 0; h < H_out; h++)
            for(int w = 0; w < W_out; w++) {
               y[b][m][h][w] = 0;
               for(int c = 0;  c < C; c++)       // sum over all input feature maps
                  for(int p = 0; p < K; p++)        // KxK  filter
                     for(int q = 0; q < K; q++)
                        y[b][m][h][w] += x[b][c][h + p][w + q] * k[m][c][p][q];
            }
}
```

2. List whole program execution time

133.67user 4.39system 2:07.99elapsed 107%CPU (0avgtext+0avgdata 5951716maxresident)
k
0inputs+1464outputs (0major+2266968minor)pagefaults 0swaps

3. List Op Times

Op Time: 21.634391
Op Time: 102.079140

4. Check correctness on the full data size of 10000

✻ Running python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 21.515465
Op Time: 102.543038
Correctness: 0.8171 Model: ece408

Milestone3

1. Implement a GPU convolution

a. dataset sizes = 100

```
✱ Running python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000610
Op Time: 0.001632
Correctness: 0.85 Model: ece408
```

b. dataset sizes = 1000

```
✱ Running python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.006196
Op Time: 0.016092
Correctness: 0.827 Model: ece408
```

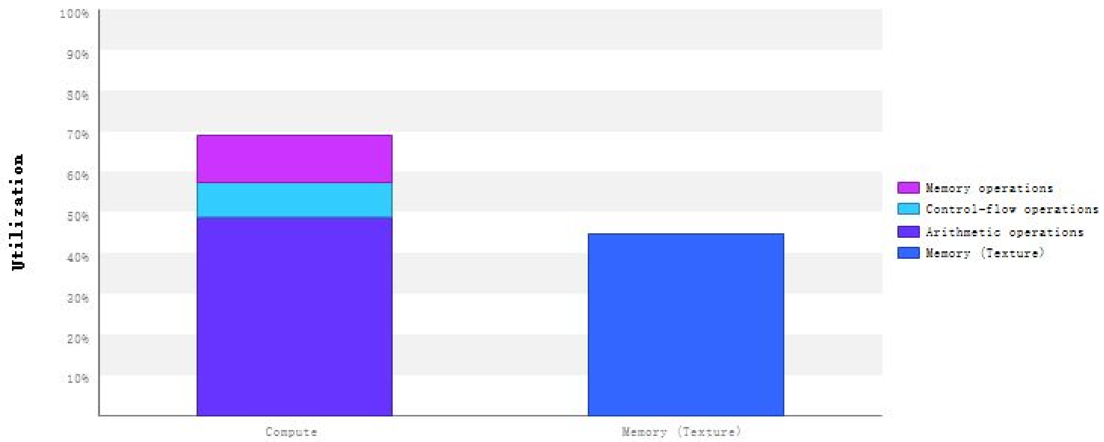
c. dataset sizes = 10000

```
* Running python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.064463
Op Time: 0.149743
Correctness: 0.8171 Model: ece408
```

2. Demonstrate nvprof profiling execution

In milestone 3, we just use a kernel and simply mapping CPU computation into GPU computation, so its performance is not not good. We will try to do some optimizations in milestone 4. Here are the performance report for our kernel:

a. kernel perfomance limiter



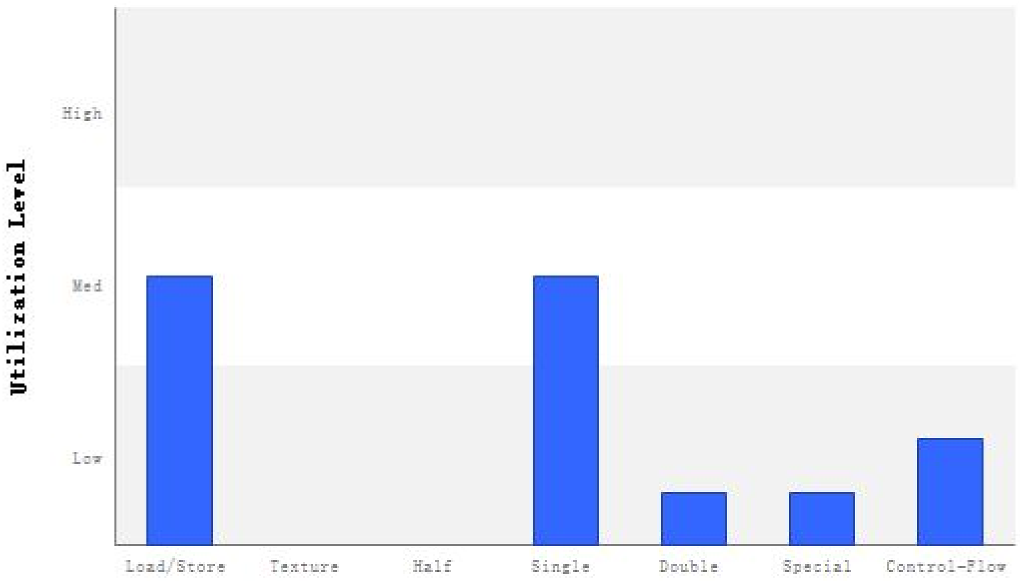
This kernel exhibits low compute throughput and memory bandwidth utilization, which are below 60%. Its performance is most likely limited by the latency of arithmetic or memory operations.

b. resource utilization

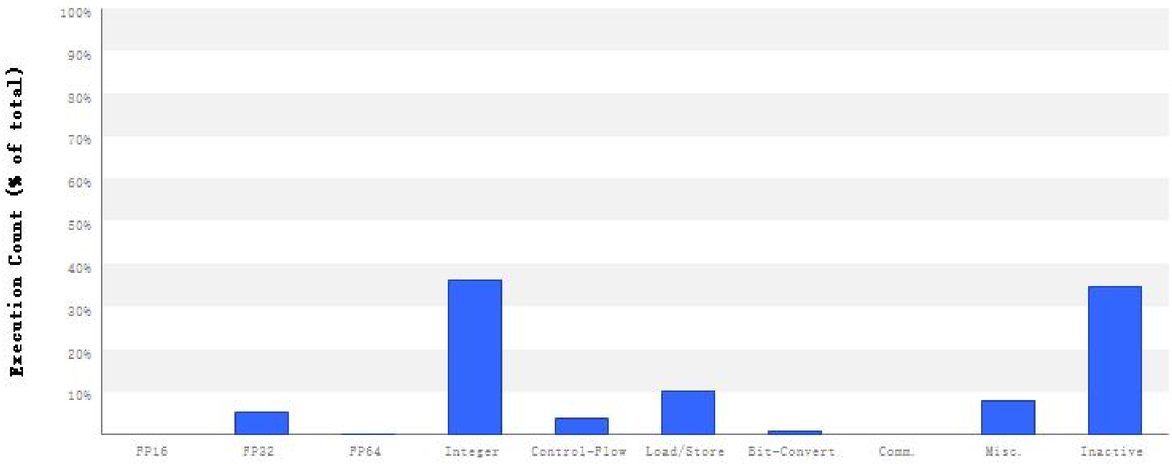
Variable	Achieved	Theoretical	Device Limit	Grid Size: [100,12,9] (10800 blocks)Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	53.62	64	64	
Active Threads		2048	2048	
Occupancy	83.8%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		32	65536	
Registers/Block		32768	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	

It shows that the kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU.

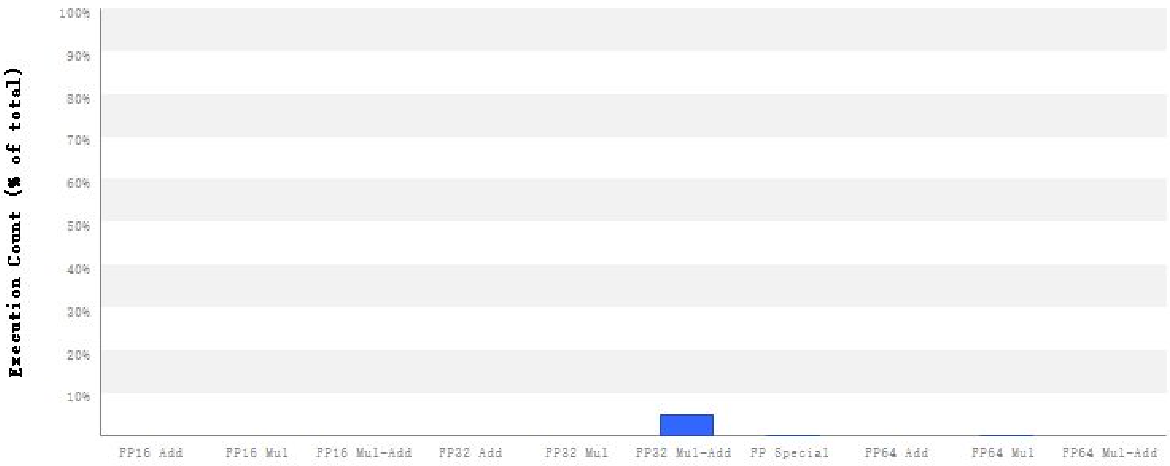
c. **kernel compute**



fuction unit utiliaztion



instruction excecution counts



floating-point operatons count

It is reported that the warp execution efficiency for these kernel is 71.9% if predicated instructions are not taken into account and the kernel's not predicated off warp execution efficiency of 65.1% is less than 100% due to divergent branches and predicated instructions.

d. kernel memory

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	1065490	62.807 GB/s	
Writes	831616	49.021 GB/s	
Total	1897106	111.828 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	53219950	3,137.136 GB/s	
Global Stores	831600	49.02 GB/s	
Texture Reads	24491449	5,774.753 GB/s	
Unified Total	78542999	8,960.909 GB/s	Idle Low Medium High Max
Device Memory			
Reads	69272	4.083 GB/s	
Writes	859137	50.643 GB/s	
Total	928409	54.727 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	294.733 kB/s	Idle Low Medium High Max

As the table shows, our kernel do not use any shared memory. The utilization of L2 cache and device memory is low relative to the maximum throughput supported by the corresponding memory.

e. divergent excecution

Divergence = 22.9% [79200 divergent executions out of 345600 total executions]

The report indicates that divergent executions in our kernel account for 22.9% of total excecutions. Therefore, divergent branches lower warp execution efficiency, which leads to inefficient use of the GPU's compute resources. That's what we need to improve in milestone 4.

Milestone4

This is the original kernel without any optimization, here for reference:

```
* Running python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.064463
Op Time: 0.149743
Correctness: 0.8171 Model: ece408
```

Then, we try 3 optimizations : ***Unrolling***, ***Shared Memory convolution*** **and** ***Weight matrix (kernel values) in constant memory***. Our analysis are as follows with the help of NVVP:

1. Optimization 1 : Unrolling

```
* Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.053189
Op Time: 0.094711
Correctness: 0.8171 Model: ece408
```

Here, we optimized the kernel by convolution loop unrolling, using "#pragma unroll". The op time are as follows:

Op Time: 0.053189

Op Time: 0.094711

As we can see, this kernel is faster than the original one.

NVVP Analysis:

```
Compute
  55.9% mxnet::op::forward_kernel_unroll2(float*, float const *, float const *, int, int, int, int, int, int)
  19.0% mxnet::op::forward_kernel_unroll1(float*, float const *, float const *, int, int, int, int, int, int)
  10.4% volta_sgemm_64x32_sliced1x4_tn
  4.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu,...
  3.5% void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTens...
  2.6% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu,...
  2.2% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, i...
  1.3% volta_sgemm_128x64_tn
  0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu,...
  0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, ...
  0.2% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, flo...
```

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, ms...	2	2.128 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Pla...	1	4.223 µs	21	1024	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, ms...	14	4.594 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, ms...	1	4.672 µs	25	0	0
volta_sgemm_128x64_tn	1	32.32 µs	122	12800	0
void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1,...	2	42.063 µs	22	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, ms...	2	50.719 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::ma...	1	52.544 µs	48	0	3872
volta_sgemm_64x32_sliced1x4_tn	1	252.605 µs	138	26624	0
mxnet::op::forward_kernel_unroll1(float*, float const *, float const *, in...	1	460.443 µs	31	0	0
mxnet::op::forward_kernel_unroll2(float*, float const *, float const *, in...	1	1.35851 ms	32	0	0

Our Analysis:

Since the kernel size is fixed(7), we can use loop unrolling to transfer dynamic loop into static loop. Loop unrolling is effective due to the memory access in computer. For example, "for(int i = 0; i < 2; i++) b[i] += 1" is slower than "b[0] += 1; b[1] += 1;". This is because in the former version, variable i increases 2 times and assign to b 2 times, whereas the latter version only increase i by 2.

2. Optimization 2 : Shared Memory convolution

```
✱ Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.052314
Op Time: 0.200539
Correctness: 0.8171 Model: ece408
```

Then, we tried to optimize the kernel by shared memory convolution. The op time are as follows:

Op Time: 0.052314

Op Time: 0.200539

As we can see, this kernel is also faster than the original one.

NVVP Analysis:

Compute
- 66.4% mxnet::op::forward_kernel_shared2(float*, float const *, float const *, int, int, int, int, int, int)
- 15.1% mxnet::op::forward_kernel_shared1(float*, float const *, float const *, int, int, int, int, int, int)
- 7.8% volta_sgemm_64x32_sliced1x4_tn
- 3.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, flo...
- 2.5% void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, fl...
- 1.9% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, flo...
- 1.6% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool...
- 1.0% volta_sgemm_128x64_tn
- 0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, flo...
- 0.1% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, msha...
- 0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, flo...

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<...	1	4.288 µs	21	1024	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<m...	1	4.576 µs	25	0	0
volta_sgemm_128x64_tn	1	32.32 µs	122	12800	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, ...	1	52.159 µs	48	0	3872
volta_sgemm_64x32_sliced1x4_tn	1	256.669 µs	138	26624	0
mxnet::op::forward_kernel_shared1(float*, float const *, float const *, int, int, int, int, int, int)	1	501.019 µs	40	0	980
mxnet::op::forward_kernel_shared2(float*, float const *, float const *, int, int, int, int, int, int)	1	2.19668 ms	40	0	980
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<ms...	2	2.096 µs	16	0	0
void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::t...	2	42.16 µs	22	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<m...	2	50.415 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<m...	14	4.541 µs	16	0	0

Our Analysis:

For this optimization, we load tiles from X[n, c,…] into shared memory which could be reused for multiple times when do the convolution with W. Thus, the kernel could perform more efficient since it reduces the time that costs to read from global memory and write back to global memory. But the optimization is limited by introducing calculation overhead, like tiling address calculations, so the improvement is not as great as what we thought it was at the very beginning.

3. Optimization 3 : Weight matrix in constant memory

```
✱ Running python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.035808
Op Time: 0.092208
Correctness: 0.8171 Model: ece408
```

Finally, for optimization 3, we tried to put weight matrix in constant memory and make kernel fusion with optimization 1 and optimization 2. The op time are as follows:

Op Time: 0.035808

Op Time: 0.092208

As we can see, this kernel is way more faster than all of other kernels.

NVVP Analysis:

- Compute
  - 49.6% mxnet::op::forward_kernel_unroll2(float*, float const *, float const *, int, int, int, int, int, int)
  - 18.9% mxnet::op::forward_kernel_unroll1(float*, float const *, float const *, int, int, int, int, int, int)
  - 13.0% volta_sgemm_64x32_sliced1x4_tn
  - 4.4% void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, c...
  - 5.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>, m...
  - 3.3% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, m...
  - 2.7% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnT...
  - 1.7% volta_sgemm_128x64_tn
  - 0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, m...
  - 0.2% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshadow::expr::Pl...
  - 0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, m...

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor...	2	2.096 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu...	1	4.352 µs	21	1024	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor...	1	4.544 µs	25	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor...	14	4.61 µs	16	0	0
volta_sgemm_128x64_tn	1	32.192 µs	122	12800	0
void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float...	2	42.959 µs	22	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor...	2	50.799 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanProp...	1	52.031 µs	48	0	3872
volta_sgemm_64x32_sliced1x4_tn	1	253.341 µs	138	26624	0
mxnet::op::forward_kernel_unroll1(float*, float const *, float const *, int, int, int, int, int, int)	1	367.803 µs	31	0	0
mxnet::op::forward_kernel_unroll2(float*, float const *, float const *, int, int, int, int, int, int)	1	966.453 µs	31	0	0

Our Analysis:

For this optimization, we copied the weight matrix into constant memory. So during the kernel executes convolution multiplications, it does not need to read value of weight in global memory and instead gets in constant memory. By this way, the kernel reduces global memory accesses and performs way better than other kernel.