

## Milestone 4

### 1. Three Optimization

#### a. Shared Memory convolution

We utilize shared memory while loading the batch input image.

#### b. Weight matrix (kernel values) in constant memory

We first count the maximum element in the kernel in two forward layers and then declared the constant memory size for the kernel.

#### c. Sweeping various parameters to find the best values (block sizes, amount of thread coarsening)

We use different size of TILE\_WIDTH: 32, 16, 8 to examine the performance. We found out that when TILE\_WIDTH is 8, we can have the lowest percentage of control divergence.

### 2. Demonstrate nvprof profiling the execution

The bottleneck of our kernel is computation.

For computation analysis:

The average control divergence for forward layer 1 is around 14% and for layer 2 is around 28%, we use TILE\_WIDTH = 8 to achieve lower control divergence. Next, we'll try to set different TILE\_WIDTH respectively.

#### Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

*Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.*

▼ Line / File: new-forward.cuh - /mxnet/src/operator/custom

60	Divergence = 15.4% [ 3000000 divergent executions out of 19440000 total executions ]
81	Divergence = 9.5% [ 12960000 divergent executions out of 136080000 total executions ]
81	Divergence = 14.6% [ 19800000 divergent executions out of 136080000 total executions ]
81	Divergence = 14.6% [ 19800000 divergent executions out of 136080000 total executions ]
81	Divergence = 9% [ 12240000 divergent executions out of 136080000 total executions ]
81	Divergence = 14% [ 19080000 divergent executions out of 136080000 total executions ]
81	Divergence = 14% [ 19080000 divergent executions out of 136080000 total executions ]
81	Divergence = 14.6% [ 19800000 divergent executions out of 136080000 total executions ]
95	Divergence = 15.4% [ 3000000 divergent executions out of 19440000 total executions ]

### 🔗 Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.


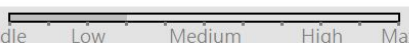
*Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.*

[More...](#)

Line / File	new-forward.cuh - \mxnet\src\operator\custom
60	Divergence = 31.2% [ 28800000 divergent executions out of 92160000 total executions ]
81	Divergence = 17.9% [ 115200000 divergent executions out of 645120000 total executions ]
81	Divergence = 28.1% [ 181440000 divergent executions out of 645120000 total executions ]
81	Divergence = 25.9% [ 167040000 divergent executions out of 645120000 total executions ]
81	Divergence = 28.1% [ 181440000 divergent executions out of 645120000 total executions ]
81	Divergence = 28.1% [ 181440000 divergent executions out of 645120000 total executions ]
81	Divergence = 15.6% [ 100800000 divergent executions out of 645120000 total executions ]
81	Divergence = 17.9% [ 115200000 divergent executions out of 645120000 total executions ]
97	Divergence = 31.2% [ 2400000 divergent executions out of 7680000 total executions ]

For memory bandwidth analysis:

Since we apply shared memory in our implementation, as the figure shown below, the unified total is 5874 GB in forward layer 1, and 5858 GB for forward layer 2, compared to 13000 GB in milestone 3.

Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	4138365186	1,621.865 GB/s	
Global Stores	118800000	46.559 GB/s	
Texture Reads	2682714721	4,205.526 GB/s	
Unified Total	6939879907	5,873.95 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	22576583712	1,963.357 GB/s	
Global Stores	44550000	3.874 GB/s	
Texture Reads	11185758669	3,891.047 GB/s	
Unified Total	33806892381	5,858.278 GB/s	

For latency analysis:

Occupancy can be improved by reducing the register. However, the register is used for if-else condition, we cannot remove them. The next solution is using matrix multiplication to implement convolution. By doing so, we can use fewer registers and reduce branches at the same time.

## Milestone 3

### 1. Correctness and timing with 3 different dataset sizes

```
* Running /usr/bin/time python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000459
Op Time: 0.001642
Correctness: 0.85 Model: ece408
4.33user 2.53system 0:04.54elapsed 151%CPU (0avgtext+0avgdata 2629084maxresid
ent)k
0inputs+4592outputs (0major+604222minor)pagefaults 0swaps
```

```
* Running /usr/bin/time python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.004393
Op Time: 0.016343
Correctness: 0.827 Model: ece408
4.21user 2.44system 0:04.41elapsed 151%CPU (0avgtext+0avgdata 2634592maxresident)k
0inputs+0outp
uts (0major+605784minor)pagefaults 0swaps
```

```
* Running /usr/bin/time python m3.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.044400
Op Time: 0.149469
Correctness: 0.8171 Model: ece408
4.21user 2.58system 0:04.79elapsed 141%CPU (0avgtext+0avgdata 2823492maxresident)k
0inputs
+0outputs (0major+703643minor)pagefaults 0swaps
```

## 2. Demonstrate nvprof profiling the execution

Our kernel is bounded by memory bandwidth and computation, as shown in below figure. So in this report, we will focus on the analyses of these two parts.

For computation analysis:

**Divergent Branches**

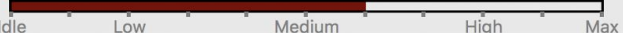
Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. [More...](#)

▼ Line / File	new-forward.cuh - /mxnet/src/operator/custom
25	Divergence = 46.9% [ 36000 divergent executions out of 76800 total executions ]

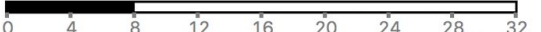

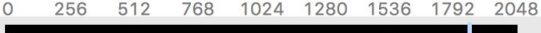

The compute divergence is about 47%. In the implementation, the if-else condition for boundary checking causes divergence. According to our computation, there are 12 warps out of 32 warps will suffer from branch divergence, which is around 38% divergence. However, the calculated number is not consistent with the analysis, we are still trying to figure out what causes the discrepancy. A simple solution to mitigate the divergence is to adjust the TILE\_WIDTH, we will try this idea in milestone 4.

For memory bandwidth analysis:

Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	234568449	5,098.379 GB/s	
Global Stores	332100	7.218 GB/s	
Texture Reads	90425675	7,861.66 GB/s	
Unified Total	325326224	12,967.257 GB/s	

We can see that the memory utilization of global memory read is pretty high. This is because that, in our kernel implementation, we simply read the input directly from the global memory without loading it into shared memory first. A simple optimization to resolve this is loading input tiles into shared memory to enable data reuse, which we will do in milestone 4.

Although the analysis shows our kernel is fine with latency, we still show the results here for completeness.

Occupancy Per SM				
Active Blocks		8	32	
Active Warps	57.94	64	64	
Active Threads		2048	2048	
Occupancy	90.5%	100%	100%	

We can see clearly that the occupancy of SMs is pretty high, so there should be plenty of warps to switch around for latency hiding.

## Milestone 2

---

1. Whole Program Execution Time

The whole program execution time is 181.21 seconds.

2. List Op Times

Op1: 25.45s, Op2: 151.55 s

```
* Running /usr/bin/time python m2.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 25.453350
Op Time: 151.554303
Correctness: 0.8171 Model: ece408
187.13user 4.34system 3:01.21elapsed 105%CPU (0avgtext+0avgdata 5952880maxresident)k
0inputs+0outputs (0major+2266839minor)pagefa
ults 0swaps
```

# Milestone 1

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

**36.91%** [CUDA memcpy HtoD]

**22.46%** volta\_scudnn\_128x32\_relu\_interior\_nn\_v1

**20.94%** 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)

**7.31%** 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\*)

**7.31%** volta\_sgemm\_128x128\_tn

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

**39.11%** cudaStreamCreateWithFlags

**36.01%** cudaMemGetInfo

**20.84%** cudaFree

3. An explanation of the difference between kernels and API calls.

Kernels are implemented by the programmers to realize a specific function/computation  
API is only a basic function which let the programmers communicate with GPU to transfer data, memory allocation...etc.

4. Show output of rai running MXNet on the CPU

```
* Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
19.66user 3.93system 0:13.38elapsed 176%CPU (0avgtext+0avgdata 5956652maxresident)k
0inputs+2856outputs (0major+1585294minor)pagefaults 0swaps
```

5. List program run time

The running time for the CPU version is 13.38 seconds



6. Show output of rai running MXNet on the GPU

```
* Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
4.29user 2.80system 0:04.66elapsed 2152%CPU (0avgtext+0
avgdata 2832216maxresident)k
0inputs+4568outputs (0major+702735minor)pagefaults 0swaps
```

7. List program run time

The running time for the GPU version is 4.66 seconds