# MileStone4 Report
# ECE408

Team : lakersfirst
School affiliation: campus students
hz43 (Haoyuan Zhang) xiangl14 (Xiang Li) xic10 (Xi Chen)

## Optimization1: Weight matrix in constant memory

Op Time: 0.026449
Op Time: 0.082417
Correctness: 0.7653 Model: ece408

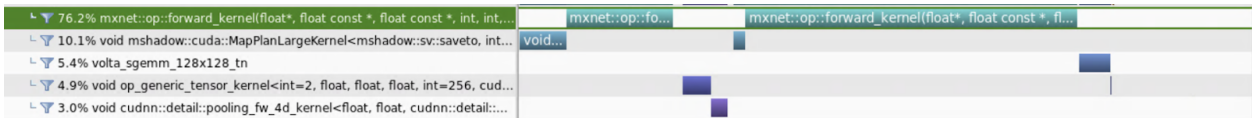
As shown in the below images, duration time of forward kernel for layer 1 is 27ms and duration time of forward kernel for layer 2 is 84ms.
For milestone 3, it is 30 ms and 93 ms.
So the performance is improved.

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

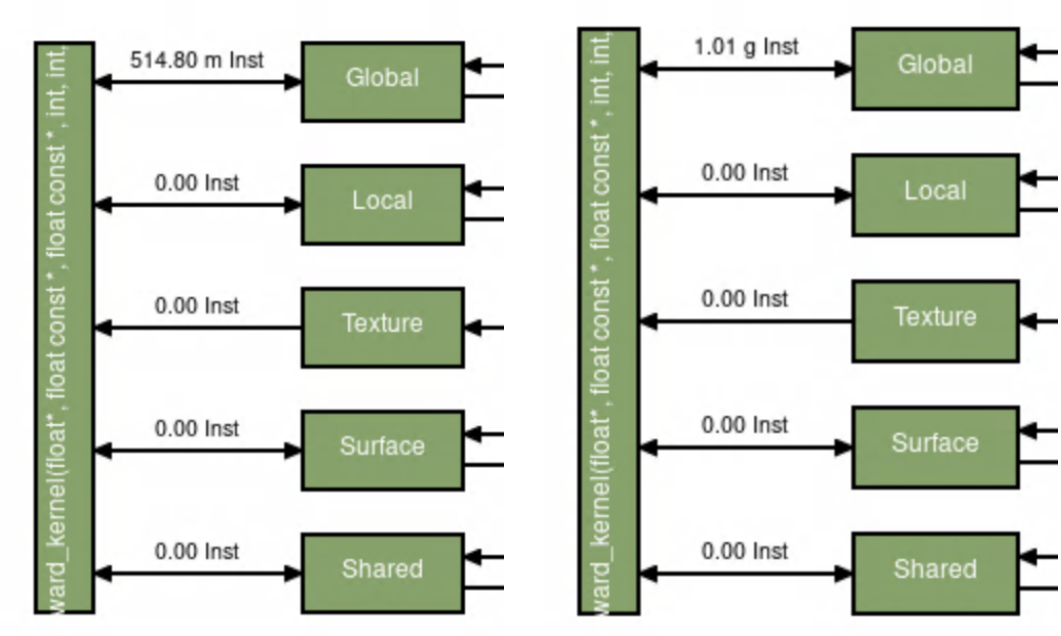
| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 5.04247 s (5,042,471,209 ns) |
| End | 5.06962 s (5,069,615,594 ns) |
| Duration | 27.14438 ms (27,144,385 ns) |
| Stream | Default |
| Grid Size | [ 10000,12,25 ] |
| Block Size | [ 16,16,1 ] |
| Registers/Thread | 32 |
| Shared Memory/Block | 0 B |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 100% |
| ▼ Shared Memory Configuration | |
| Shared Memory Executed | 0 B |
| Shared Memory Bank Size | 4 B |

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

| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 5.08832 s (5,088,321,425 ns) |
| End | 5.17327 s (5,173,268,633 ns) |
| Duration | 84.94721 ms (84,947,208 ns) |
| 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 Executed | 0 B |
| Shared Memory Bank Size | 4 B |



The following two figures show memory access. Left one is for this implementation right one is for milestone3. Using constant memory to store weight matrix, the global memory access is reduced by half.




# Optimization2: Shared Memory convolution

Op Time: 0.039011
Op Time: 0.117171
Correctness: 0.7653 Model: ece408

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

| Queued | n/a |
|---|---|
| Submitted | n/a |
| Start | 121.70234 s (121,702,336,613 ns) |
| End | 121.74128 s (121,741,284,726 ns) |
| Duration | 38.94811 ms (38,948,113 ns) |
| Stream | Default |
| Grid Size | [ 10000,12,25 ] |
| Block Size | [ 16,16,1 ] |
| Registers/Thread | 40 |
| Shared Memory/Block | 1.66 KiB |
| Launch Type | Normal |
| ▾ Occupancy | |
| Theoretical | 75% |
| ▾ Shared Memory Configuration | |
| Shared Memory Executed | 16 KiB |
| Shared Memory Bank Size | 4 B |

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

| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 121.75957 s (121,759,567,399 ns) |
| End | 121.87668 s (121,876,678,455 ns) |
| Duration | 117.11106 ms (117,111,056 ns) |
| Stream | Default |
| Grid Size | [ 10000,24,4 ] |
| Block Size | [ 16,16,1 ] |
| Registers/Thread | 40 |
| Shared Memory/Block | 1.66 KiB |
| Launch Type | Normal |
| ▼ Occupancy | |
| Theoretical | 75% |
| ▼ Shared Memory Configuration | |
| Shared Memory Executed | 16 KiB |
| Shared Memory Bank Size | 4 B |

As shown in the images, duration time of forward kernel for layer 1 is 38ms and duration time of forward kernel for layer 2 is 117ms.For milestone 3, it is 30 ms and 93 ms.
So the performance gets worse.

| | Transactions | Bandwidth | Utilization |
|---|---|---|---|
| Shared Memory | | | |
| Shared Loads | 1834797428 | 6,132.986 GB/s | |
| Shared Stores | 100103836 | 334.607 GB/s | |
| Shared Total | 1934901264 | 6,467.593 GB/s | Idle Low Medium High Max |
| L2 Cache | | | |
| Reads | 150119744 | 125.447 GB/s | |
| Writes | 95040022 | 79.42 GB/s | |
| Total | 245159766 | 204.868 GB/s | Idle Low Medium High Max |
| Unified Cache | | | |
| Local Loads | 0 | 0 B/s | |
| Local Stores | 0 | 0 B/s | |
| Global Loads | 232381619 | 194.189 GB/s | |
| Global Stores | 95040000 | 79.42 GB/s | |
| Texture Reads | 1360606299 | 4,547.957 GB/s | |
| Unified Total | 1688027918 | 4,821.566 GB/s | Idle Low Medium High Max |
| Device Memory | | | |
| Reads | 201217478 | 168.147 GB/s | |
| Writes | 95092713 | 79.464 GB/s | |
| Total | 296310191 | 247.611 GB/s | Idle Low Medium High Max |
| System Memory [ PCIe configuration: Gen3 x8, 8 Gbit/s ] | | | |
| Reads | 0 | 0 B/s | Idle Low Medium High Max |
| Writes | 5 | 4.178 kB/s | Idle Low Medium High Max |

For this optimization, we load tiles from X[n, c,…] and W[m, c, …]into shared memory which could be reused for multiple times when do the convolution. Thus, the kernel could perform better memory utilization since it reduces the time that costs to read from global memory. However, in the loading process of input, the control divergence exists, which will effect the final running time( below is specific 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 |
|---|---|
| 60 | Divergence = 37.5% [ 9000000 divergent executions out of 24000000 total executions ] |
| 66 | Divergence = 100% [ 30000000 divergent executions out of 30000000 total executions ] |
| 66 | Divergence = 100% [ 30000000 divergent executions out of 30000000 total executions ] |
| 88 | Divergence = 16.5% [ 3960000 divergent executions out of 24000000 total executions ] |

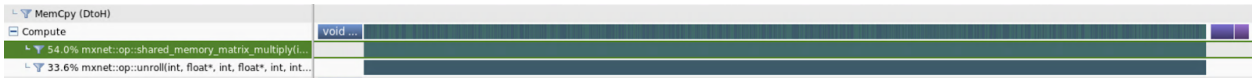
```
Line 65: for( int i=h; i<h_base+X_tile_width; i+=TILE_WIDTH)
Line 66:    for( int j=w; j<w_base+X_tile_width; j+=TILE_WIDTH)
Line 67:       X_shared[(i-h_base)*X_tile_width+(j-w_base)] = x4d(b,c,i,j);
```

Here, we get 100% divergence when loading input feature maps into shared memory. In our strategy1, we allocate shared memory with input feature map's size, which is necessarily greater than output size. In case that we launch kernel with blockSize that equals to output size, some threads will run twice in for-loop to load data while most will only load once (most threads index fail the condition "j < w_base + X_tile_width after move TILE_WIDTH forwards). Therefore, each warp will have branch divergence.

# Optimization3: Unroll + shared-memory Matrix multiply

Op Time: 0.224242
Op Time: 0.315163
Correctness: 0.7653 Model: ece408

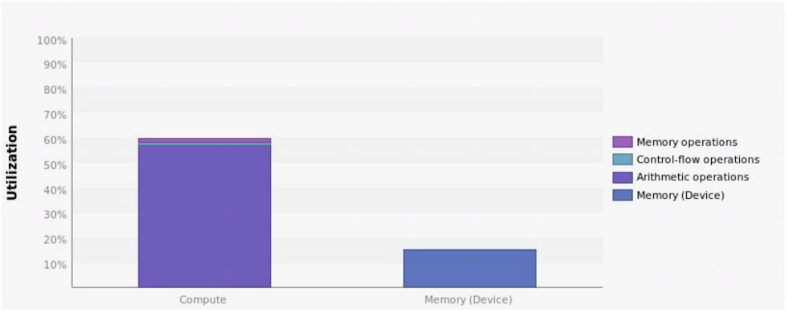
Because memory is not big enough to store X_unroll with size B*C*K*K*H_out*W_out, we have to reuse the buffer X_unroll with size C*K*K*H_out*W_out for B iterations. The decline in run-time performance is mainly due to B iterations (which will be relatively large in some cases). Instead,  using mini-batch training schema
The figure shows that kernel Unroll and kernel Matrix Multiplication are used alternately 10000 times. The performance is reduced because of the for loop.



## Unroll

**Kernel Performance Is Bound By Instruction And Memory Latency**

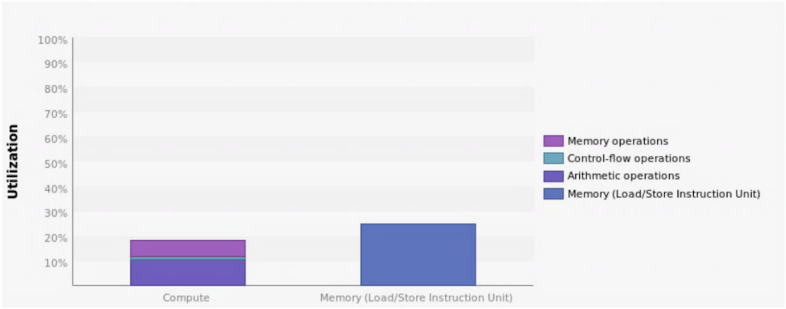
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.




## Matrix multiplication

**Kernel Performance Is Bound By Instruction And Memory Latency**

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



# MileStone3 Report
# ECE408

Team : lakersfirst
School affiliation: campus students
hz43 (Haoyuan Zhang) xiangl14 (Xiang Li) xic10 (Xi Chen)

## 1.Correctness and timing with 3 different data size

**(1) Image Set Size = 100**

Op Time: 0.000420
Op Time: 0.002142
Correctness: 0.76 Model: ece408
5.06user 3.13system 0:05.72elapsed

**(2) Image Set Size = 1000**

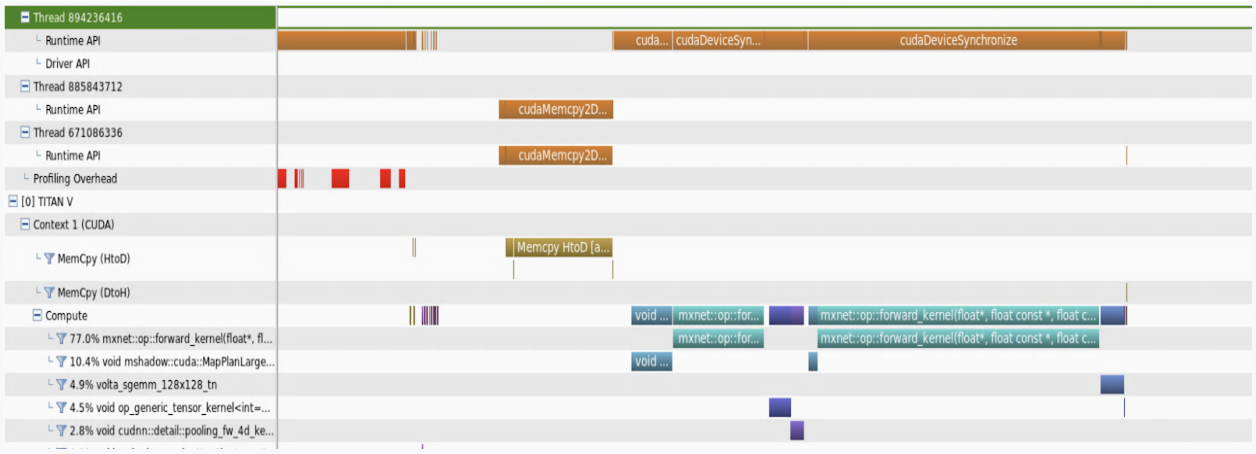
Op Time: 0.002681
Op Time: 0.009407
Correctness: 0.767 Model: ece408
4.95user 2.90system 0:04.36elapsed

**(3) Image Set Size = 10000**

Op Time: 0.030741
Op Time: 0.093671
Correctness: 0.7653 Model: ece408
5.34user 2.82system 0:05.78elapsed

## 2. Demonstrate nvprof profiling the execution

**(1) Execution Timeline**



| mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int) | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 4.79226 s (4,792,262,008 ns) |
| End | 4.82248 s (4,822,475,513 ns) |
| Duration | 30.21351 ms (30,213,505 ns) |
| 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 Executed | 0 B |
| Shared Memory Bank Size | 4 B |

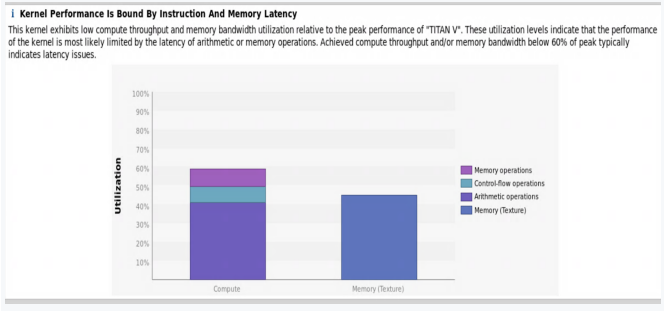
Figure. Property of first-time running of forward_kernel

| mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int) | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 4.84036 s (4,840,357,903 ns) |
| End | 4.93407 s (4,934,074,739 ns) |
| Duration | 93.71684 ms (93,716,836 ns) |
| 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 Executed | 0 B |
| Shared Memory Bank Size | 4 B |

Figure. Property of second-time running of forward_kernel

According to the timeline visualization result, forward_kernel is executed twice in the neural network.In our GPU implementation, we use Tile_Size as 16. First time with Grid size (10000,12,25), with 12 output feature maps and 25 blocks. Second time with Grid size (10000,24,4), with 24 output feature maps and 4 blocks.

### (2) Kernel analysis



### (3) Control Divergence

**Low Warp Execution Efficiency**

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The warp execution efficiency for these kernels is 84.8% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 76.6% is less than 100% due to divergent branches and predicated instructions.

*Optimization: Reduce the amount of intra-warp divergence and predication in the kernel.* More...

**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 |
|---|---|
| 43 | Divergence = 16.5% [ 3960000 divergent executions out of 24000000 total executions ] |

We know that when threads in the same warp act different behaviors, we'll have control divergence. As stated in the "Divergent Branches" metrix, we can trace back to our code and see that divergence happens in boundary check (in line 43). That is, 16.5% of the thread don't participate in the actual convolution process in that they are out of  shape boundary of output feature maps.

Reference: Line 43:

```
if (h < H_out && w < W_out){
```

### (4) Latency Analysis

**Occupancy Is Not Limiting Kernel Performance**

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

| Variable | Achieved | Theoretical | Device Limit | Grid Size: [ 10000,12,25 ] (3000000 blocks)Block Size: [ 16,16,1 ] (256 threads) |
|---|---|---|---|---|
| Occupancy Per SM | | | | |
| Active Blocks | | 8 | 32 | 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 |
| Active Warps | 52.01 | 64 | 64 | 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 |
| Active Threads | | 2048 | 2048 | 0 256 512 768 1024 1280 1536 1792 2048 |
| Occupancy | 81.3% | 100% | 100% | 0% 15% 30% 45% 60% 75% 90% 100% |
| Warps | | | | |
| Threads/Block | | 256 | 1024 | 0 128 256 384 512 640 768 896 1024 |
| Warps/Block | | 8 | 32 | 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 |
| Block Limit | | 8 | 32 | 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 |
| Registers | | | | |
| Registers/Thread | | 32 | 65536 | 0 8192 16384 24576 32768 40960 49152 57344 65536 |
| Registers/Block | | 8192 | 65536 | 0 8k 16k 24k 32k 40k 48k 56k 64k |
| Block Limit | | 8 | 32 | 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 |
| Shared Memory | | | | |
| Shared Memory/Block | | 0 | 98304 | 0 16k 32k 48k 64k 80k 96k |
| Block Limit | | 0 | 32 | 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 |

## (5) Memory Analysis

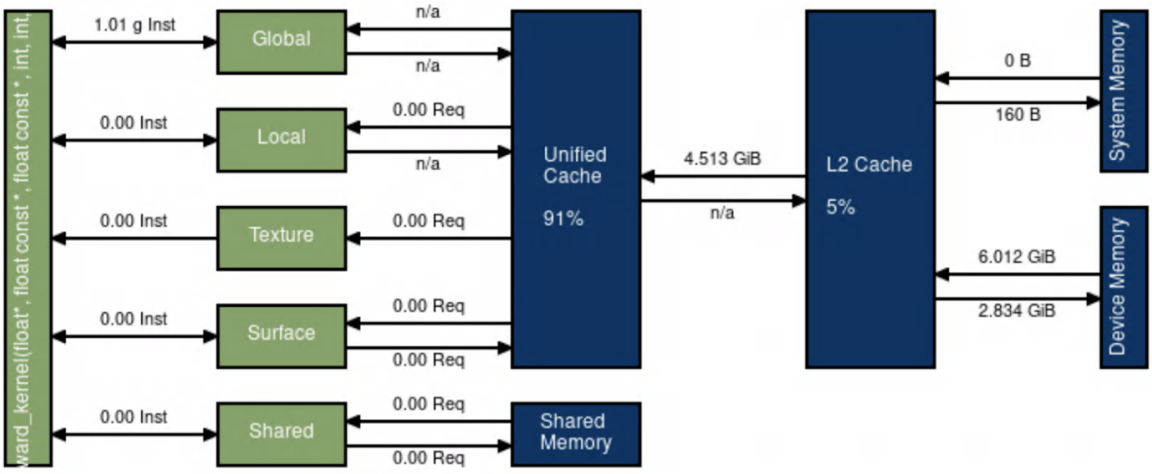
**Memory Bandwidth And Utilization**

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

| | 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 | 151499015 | 159.187 GB/s | |
| Writes | 95040498 | 99.864 GB/s | |
| Total | 246539513 | 259.051 GB/s | Idle Low Medium High Max |
| Unified Cache | | | |
| Local Loads | 0 | 0 B/s | |
| Local Stores | 0 | 0 B/s | |
| Global Loads | 2969929768 | 3,120.65 GB/s | |
| Global Stores | 95040000 | 99.863 GB/s | |
| Texture Reads | 1285463423 | 5,402.797 GB/s | |
| Unified Total | 4350433191 | 8,623.31 GB/s | Idle Low Medium High Max |
| Device Memory | | | |
| Reads | 201735040 | 211.973 GB/s | |
| Writes | 95094768 | 99.921 GB/s | |
| Total | 296829808 | 311.894 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 | 5.253 kB/s | Idle Low Medium High Max |

**Memory Statistics**

The following chart shows a summary view of the memory hierarchy of the CUDA programming model. The green nodes in the diagram depict logical memory space whereas blue nodes depicts actual hardware unit on the chip. For the various caches the reported percentage number states the cache hit rate; that is the ratio of requests that could be served with data locally available to the cache over all requests made.
The links between the nodes in the diagram depict the data paths between the SMs to the memory spaces into the memory system. Different metrics are shown per data path. The data paths from the SMs to the memory spaces report the total number of memory instructions executed, it includes both read and write operations. The data path between memory spaces and "Unified Cache" or "Shared Memory" reports the total amount of memory requests made (read or write). All other data paths report the total amount of transferred memory in bytes.



# MileStone2 Report
# ECE408

Team : lakersfirst
School affiliation: campus students
hz43 (Haoyuan Zhang) xiangl14 (Xiang Li) xic10 (Xi Chen)

https://github.com/TonyLidiang/ECE408/invitations

**1.List of all kernels that collectively consume more than 90% of the program time**

| Name | Time(%) | Time | Calls | Avg | Min | Max |
|---|---|---|---|---|---|---|
| [CUDA memcpy HtoD] | 31.68% | 35.741ms | 20 | 1.7871ms | 1.0560us | 33.422ms |
| volta_scudnn_128x64_relu_interior_nn_v1 | 17.57% | 19.821ms | 1 | 19.821ms | 19.821ms | 19.821ms |
| volta_gcgemm_64x32_nt | 16.99% | 19.168ms | 4 | 4.7921ms | 4.7885ms | 4.7966ms |
| fft2d_c2r_32x32 | 8.63% | 9.7399ms | 4 | 2.4350ms | 2.0336ms | 3.1590ms |
| volta_sgemm_128x128_tn | 7.71% | 8.7011ms | 1 | 8.7011ms | 8.7011ms | 8.7011ms |
| op_generic_tensor_kernel | 6.50% | 7.3307ms | 2 | 3.6653ms | 25.952us | 7.3047ms |
| fft2d_r2c_32x32 | 6.38% | 7.1967ms | 4 | 1.7992ms | 1.4356ms | 2.2607ms |
| cudnn::detail::pooling_fw_4d_kernel | 3.90% | 4.4044ms | 1 | 4.4044ms | 4.4044ms | 4.4044ms |

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

| API Calls | Time | Time(%) | Calls | Avg |
|---|---|---|---|---|
| cudaStreamCreateWithFlags | 3.68363s | 43.57% | 22 | 167.44ms |
| cudaMemGetInfo | 2.68858s | 31.80% | 24 | 112.02ms |
| cudaFree | 1.79272s | 21.21% | 19 | 94.354ms |

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

The kernel is the device code(run in GPU) that marked with CUDA keywords for data-parallel functions and kernels are usually written by ourselves.
API calls are calls made by the host code into the CUDA driver or runtime libraries, such as cudaMalloc(), cudaFree() and cudaMemcpy() functions.

**4. Output of rai running MXNet on the CPU**

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

### 5. Program run time on CPU

17.10user 4.86system 0:09.07elapsed

**6. Output of rai running MXNet on the GPU**

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

**7. Program run time on GPU**

5.11user 3.41system 0:04.72elapsed

**8. CPU Program 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 each image in batch
      for(int m = 0;  m < M;  m++)             // for each output feature map
         for(int h = 0; h < H_out; h++)          // for each output element
            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 (channels)
                    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];
            }
}
```

### 9.Whole program execution time

M2.1 Output:
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 11.806489
Op Time: 60.063094
Correctness: 0.7653 Model: ece408
85.46user 7.61system 1:15.77elapsed 122%CPU (0avgtext+0avgdata 6044368maxresident)k
0in
puts+0outputs (0major+2310434minor)pagefaults 0swaps

Whole Program run time: User: 85.46; Sys: 7.61 Elapsed 1:15.77

**10. Op Time**

Op Time: 11.806489
Op Time: 60.063094