ECE 408 Course Project Report

Team Name: cudnn_think_of_one
School Affiliation: UIUC

Team Members:
Ayush Agarwal (ayusha4)
Shivam Bharuka (bharuka2)
Vandana Kulkarni (vandana2)

Milestone 4

Optimization 1: Unrolling and Matrix-Multiplication

The output matrix is created using simple matrix multiplication between the weight matrix and the input image matrix. We implement two kernels: (i) unroll each image in the batch, X tensor, (ii) matrix mutliplication between the weight matrix and unrolled image matrix.

Function	**Constant**
Number of images in the input	B
Number of output feature maps	M
Number of input channels	C
Kernel dimension	K * K
Height of the input image	H
Width of the input image	W
Height of the output feature	H_out
Width of the output feature	W_out

Width of the weight matrix = C * K * K
Height of the weight matrix = M

Width of the unrolled X matrix = H_out * W_out
Height of the unrolled X matrix = C * K * K

Kernel 1 (Unroll Image Data):

```
__global__ void forward_kernel_unroll(const float* x, float* unroll_x,
    const int H, const int W, const int B, const int C, const int K,
    const int W_out, const int matrixHeight, const int matrixWidth) {

    #define x4d(b,m,h,w) x[(b) * (C * H * W) + (m) * (H * W) + (h) * (W) + w]
    #define y4d(m,h,w) unroll_x[(m) * (matrixHeight * matrixWidth) + (h) *
(matrixWidth) + w]

    const int threadIndex = blockIdx.x * blockDim.x + threadIdx.x;

    if (threadIndex < C * matrixWidth) {
        const int row = (threadIndex % matrixWidth) / W_out;
        const int column = (threadIndex % matrixWidth) % W_out;

        for (int i = 0; i < K; ++i) {
            for (int j = 0; j < K; ++j) {
                y4d(blockIdx.y, (threadIndex / matrixWidth * K * K) + (i * K) + j,
row * W_out + column) = x4d(blockIdx.y, threadIndex / matrixWidth, row + i, column
+ j);
            }
        }
    }
}
```

Kernel 2 (Matrix Multiplication):

```
__global__ void matrixMultiply(float *A, float *B, float *C, int numARows,
                               int numAColumns, int numBRows,
                               int numBColumns, int numCRows,
                               int numCColumns) {
  //@@ Insert code to implement matrix multiplication here
  float value = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int column = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < numARows && column < numBColumns) {
    for (int i = 0; i < numAColumns; i++) {
      value += A[row * numAColumns + i] * B[(numBRows * numBColumns) * blockIdx.z + i *
numBColumns + column];
    }
    C[(numCRows * numCColumns) * blockIdx.z + row * numCColumns + column] = value;
  }
}
```

Host Code Snippet:

```
...
mshadow::Tensor<gpu, 3, float> unroll_x;
unroll_x.shape_ = mshadow::Shape3(matrixWidth, matrixHeight, B);
```

```
mshadow::AllocSpace(&unroll_x);

dim3 gridDim((NUM_THREADS+C*matrixWidth-1)/NUM_THREADS, B, 1);
dim3 blockDim(NUM_THREADS, 1, 1);

forward_kernel_unroll<<<gridDim, blockDim>>>(x.dptr_, unroll_x.dptr_, H, W, B, C, K, W_out,
matrixHeight, matrixWidth);

dim3 dimBlock(16, 16, 1);
    dim3 dimGrid((matrixWidth + dimBlock.x - 1) / dimBlock.x, (M + dimBlock.y - 1) /
dimBlock.y, B);
    matrixMultiply<<<dimGrid, dimBlock>>>(k.dptr_, unroll_x.dptr_, y.dptr_, M, matrixHeight,
matrixHeight, matrixWidth, M, matrixWidth);

mshadow::FreeSpace(&unroll_x);
```

Performance Assessment:

```
Running time for python mp3.1py 1000
New Inference
Op Time: 0.008297
Op Time: 0.021944
Correctness: 0.827 Model: ece408
4.06user 2.52system 0:04.31elapsed 152%CPU

NVVP:
Kernel 1 (Unroll):
Duration of kernel execution = 1.76ms + 4.21 ms = 5.97 ms
Shared Mem/Block = 0B

Kernel 2 (Matrix Multiplication):
Duration of kernel execution = 3.11 ms + 12.35 ms = 15.46 ms
Shared Mem/Block = 0B
```

This optimization does not seem to give us a lot of improvement in the performance due to the global memory reads per image pixel. We perform multiple reads during unrolling and then again during matrix multiplication. We also come to the conclusion that most of our running time is spent in matrix multiplication kernel whereas the unrolling kernel consumes minimal running time. Initially, we thought of optimizing the unroll kernel by loading raw image data in shared memory and then storing the unrolled data in global memory but due to the minimal running time of the unroll kernel, we decided against it and thought of optimizing the matrix multiplication kernel.

Optimization 2: Advanced Matrix-Multiplication

We optimized our matrix multiplication and decided to use tiling since we concluded that the maximum running time is spent in the matrix multiplication kernel.

Kernel 2 (Tiled Matrix Multiplication):

```
__global__ void matrixMultiplyShared(float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns) {
    float value = 0;
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    int column = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

    for (int i = 0; i < (TILE_WIDTH+numAColumns-1)/TILE_WIDTH; i++) {
        if (i*TILE_WIDTH+threadIdx.x<numAColumns && row<numARows)
            subTileM[threadIdx.y][threadIdx.x] = A[row*numAColumns + i*TILE_WIDTH
+threadIdx.x];
        else
            subTileM[threadIdx.y][threadIdx.x] = 0;

        if (i*TILE_WIDTH+threadIdx.y<numBRows && column<numBColumns)
            subTileN[threadIdx.y][threadIdx.x] = B[(numBRows * numBColumns) * blockIdx.z +
numBColumns * (i*TILE_WIDTH+threadIdx.y) + column];
        else
            subTileN[threadIdx.y][threadIdx.x] = 0;

        __syncthreads();

        if (row < numCRows && column < numCColumns) {
            for (int j = 0; j < TILE_WIDTH; j++)
                value += subTileM[threadIdx.y][j] * subTileN[j][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < numCRows && column < numCColumns)
        C[(numCRows * numCColumns) * blockIdx.z + numCColumns * row + column] = value;
}
```

Host Code Snippet:

```
...
dim3 gridMatrix((TILE_WIDTH+matrixWidth-1)/TILE_WIDTH, (TILE_WIDTH+M-1)/TILE_WIDTH, B);
dim3 blockMatrix(TILE_WIDTH, TILE_WIDTH, 1);

matrixMultiplyShared<<<gridMatrix, blockMatrix>>>(k.dptr_, unroll_x.dptr_, y.dptr_, M,
```

```
matrixHeight, matrixHeight, matrixWidth, M, matrixWidth);
...
```

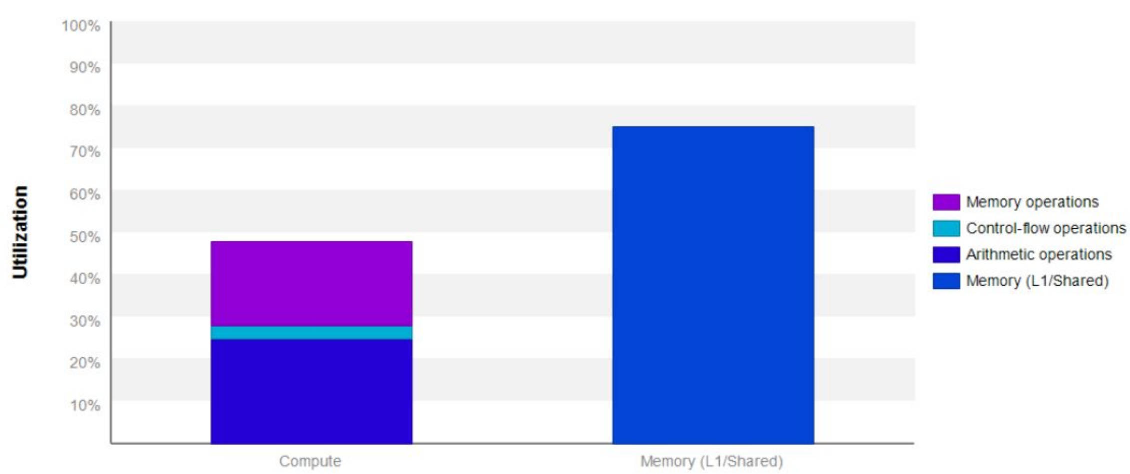
Performance Assessment:

```
Running time for python mp3.1py 1000
New Inference
Op Time: 0.008607
Op Time: 0.015810
Correctness: 0.827 Model: ece408
4.17user 2.62system 0:04.37elapsed 155%CPU
```

NVVP:

```
Kernel 1 (Unroll):
Duration of kernel execution = 1.76ms + 4.21 ms = 5.97 ms
Shared Mem/Block = 0B

Kernel 2 (Matrix Multiplication):
Duration of kernel execution = 3.94 ms + 7.52 ms = 11.46 ms
Shared Mem/Block = 8KiB
```



This optimization still does not seem to provide a lot of improvement due to the running time of the matrix multiplication kernel. Through our analysis, we see that the most running time of matrix multiplication is still spent in accessing memory rather than compute.

But when we ran these optimizations on the dataset with 10000, our implementation ran out of memory. This is because we store the unrolled matrix for all images in global memory which is not possible for 10000 images. To optimize this further, we can do two things - (i) Unroll images one by one and do the matrix multiplication, we can optimize this further by unrolling images in batches and doing the computation; (ii) Combine the kernel for matrix multiplication and unrolling and perform logical unrolling instead of allocating memory and doing physical unrolling.

We tried the first approach and unrolled images in batches and performed the computation.

NUM_IMAGES = Number of images we unroll and perform matrix multiplication.

Host Code Snippet:

```
...
mshadow::Tensor<gpu, 3, float> unroll_x;
    unroll_x.shape_ = mshadow::Shape3(matrixWidth, matrixHeight, NUM_IMAGES);
    mshadow::AllocSpace(&unroll_x);

    dim3 gridDim((NUM_THREADS+C*matrixWidth-1)/NUM_THREADS, NUM_IMAGES, 1);
    dim3 blockDim(NUM_THREADS, 1, 1);

    // Using simple matrix multiplication
    //dim3 dimBlock(16, 16, 1);
    //dim3 dimGrid((matrixWidth + dimBlock.x - 1) / dimBlock.x, (M + dimBlock.y - 1) /
dimBlock.y, NUM_IMAGES);

    // Using tiled matrix multiplication
    dim3 gridMatrix((TILE_WIDTH+matrixWidth-1)/TILE_WIDTH, (TILE_WIDTH+M-1)/TILE_WIDTH,
NUM_IMAGES);
    dim3 blockMatrix(TILE_WIDTH, TILE_WIDTH, 1);

    for (int i = 0; i < B / NUM_IMAGES; i++) {
        forward_kernel_unroll<<<gridDim, blockDim>>>(x.dptr_, unroll_x.dptr_, H, W, i, C, K,
W_out, matrixHeight, matrixWidth);
        matrixMultiplyShared<<<gridMatrix, blockMatrix>>>(k.dptr_, unroll_x.dptr_, y.dptr_,
M, matrixHeight, matrixHeight, matrixWidth, M, matrixWidth, i);
        //matrixMultiply<<<dimGrid, dimBlock>>>(k.dptr_, unroll_x.dptr_, y.dptr_, M,
matrixHeight, matrixHeight, matrixWidth, M, matrixWidth);
    }

    mshadow::FreeSpace(&unroll_x);
```

This optimization didn’t run out of memory in 10000 images with a batch size (NUM_IMAGES) of 1000.

```
✱ Running /usr/bin/time python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.006880
Op Time: 0.012503
Correctness: 0.85 Model: ece408
```

```
4.22user 2.28system 0:06.07elapsed 107%CPU (0avgtext+0avgdata
2629488maxre
sident)k
0inputs+4664outputs (0major+602604minor)pagefaults 0swaps
```

* ```
Running /usr/bin/time python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.007751
Op Time: 0.014824
Correctness: 0.827 Model: ece408
4.01user 2.47system 0:04.24elapsed 152%CPU (0avgtext+0avgdata 2653536
maxresident)k
0inputs+0outputs (0major+608832minor)pagefaults 0swaps
```

* ```
Running /usr/bin/time python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.056834
Op Time: 0.107370
Correctness: 0.8171 Model: ece408
4.41user 2.52system 0:04.61elapsed 150%CPU
```

Next, we plan to combine the kernel for matrix multiplication and unrolling and perform logical unrolling.

Optimization 3: Shared memory convolution

Shared memory convolution was one of the initial optimizations that was implemented in the GPU kernel. The motivation of loading the input image matrix into shared memory was the reuse of input elements for producing output elements within a block. If the number of global memory accesses are reduced, the total memory access time should help in improving the speed of execution.

Strategy 2:

Kernel Code:

```
__global__ void forward_kernel(float *y, const float *x, const float *k, const int
      B, const int M, const int C, const int H, const int W, const int K, int
      W_grid){

      #define y4d(b , m, h, w) y[(b) * (M * H_out * W_out) + (m) * (H_out * W_out)
      + (h) * (W_out) + w]
      #define x4d(b, c, h_plus_p, w_plus_q) x[(b) * (C * H * W) + (c) * (H * W) +
      (h_plus_p) * (W) + w_plus_q]
      #define k4d(m, c, p, q) k[(m) * (C * K * K) + (c) * (K * K) + (p) * (K) + q]
      #define kernel_shared(i, h, w) kernel[i * (K * K) + h * K + w]
      #define input_shared(i, j, k) input[i * (BLOCK_WIDTH * BLOCK_WIDTH) + j *
      BLOCK_WIDTH + k]

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

      int b = blockIdx.z;
      int m = blockIdx.x;
      int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
      int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;

      extern __shared__ float input[]; // size = C * (BLOCK_WIDTH) * (BLOCK_WIDTH)
      * sizeof(float)

      if(h >= 0 && h < H && w >= 0 && w < W)
             for (int c = 0; c < C; c++)
                    input_shared(c, threadIdx.y, threadIdx.x) = x4d(b, c, h, w);
      else
             for (int c = 0; c < C; c++)
                    input_shared(c, threadIdx.y, threadIdx.x) = 0.0;
      __syncthreads();

      float out = 0.0f;

      if (threadIdx.x < TILE_WIDTH && threadIdx.y < TILE_WIDTH){
             for (int c = 0; c < C; c++){
                    for (int p = 0; p < K; p++){
                           for (int q = 0; q < K; q++){
                                  out += k4d(m, c, p, q) * input_shared(c,
      (threadIdx.y + p), (threadIdx.x + q));
                           }
                    }
             }
```

```
            if (h < H_out && w < W_out)
                  y4d(b, m, h, w) = out;
      }

      #undef y4d
      #undef x4d
      #undef k4d
      #undef kernel_shared
      #undef input_shared
}
```

Host Code Snippet:

```
...
      dim3 gridDim(M, Y, B);
      dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH, 1);

      long size = (C * (BLOCK_WIDTH) * (BLOCK_WIDTH) * sizeof(float));
      forward_kernel<<<gridDim, blockDim, size>>>(y.dptr_, x.dptr_, k.dptr_, B, M, C, H, W,
      K, W_grid);
```

Performance Assessment:

```
Running time for python mp3.1py 100
New Inference
Op Time: 0.000576
Op Time: 0.002803
Correctness: 0.85 Model: ece408
4.39user 2.64system 0:04.58elapsed 153%CPU

Running time for python mp3.1py 1000
New Inference
Op Time: 0.005525
Op Time: 0.027520
Correctness: 0.827 Model: ece408
4.20user 2.61system 0:04.27elapsed 159%CPU

Running time for python mp3.1py 10000
New Inference
Op Time: 0.054903
Op Time: 0.256535
```

```
Correctness: 0.8171 Model: ece408
4.43user 2.79system 0:05.01elapsed 144%CPU
```

Strategy 3:

Kernel Code:

```
__global__ void forward_kernel(float *y, const float *x, const float *k, const int
      B, const int M, const int C, const int H, const int W, const int K, int
      W_grid) {

      #define y4d(b , m, h, w) y[(b) * (M * H_out * W_out) + (m) * (H_out * W_out)
      + (h) * (W_out) + w]
      #define x4d(b, c, h_plus_p, w_plus_q) x[(b) * (C * H * W) + (c) * (H * W) +
      (h_plus_p) * (W) + w_plus_q]
      #define k4d(m, c, p, q) k[(m) * (C * K * K) + (c) * (K * K) + (p) * (K) + q]
      #define kernel_shared(i, h, w) kernel[i * (K * K) + h * K + w]
      #define input_shared(i, j, k) input[i * (TILE_WIDTH * TILE_WIDTH) + j *
      TILE_WIDTH + k]

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

          int b = blockIdx.z;
          int m = blockIdx.x;
          int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
          int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;

          extern __shared__ float input[]; // size = C * (TILE_WIDTH) *
      (TILE_WIDTH) * sizeof(float)

          if(h < H && w < W)
              for (int c = 0; c < C; c++)
                  input_shared(c, threadIdx.y, threadIdx.x) = x4d(b, c, h, w);
          else
              for (int c = 0; c < C; c++)
                  input_shared(c, threadIdx.y, threadIdx.x) = 0.0;
          __syncthreads();

          float out = 0.0f;

          if (m < M && h < H_out && w < W_out){
              for (int c = 0; c < C; c++){
                  for (int p = 0; p < K; p++){
                      for (int q = 0; q < K; q++){
                           if (((threadIdx.y + p) < TILE_WIDTH) && ((threadIdx.x +
      q) < TILE_WIDTH))
```

```
                                out += k4d(m, c, p, q) * input_shared(c, (threadIdx.y
      + p), (threadIdx.x + q));
                            else
                                out += k4d(m, c, p, q) * x4d(b, c, h+p, w+q);
                        }
                    }
                }
                y4d(b, m, h, w) = out;
            }

      #undef y4d
      #undef x4d
      #undef k4d
      #undef kernel_shared
      #undef input_shared
}
```

Host Code Snippet:

```
...
      dim3 gridDim(M, Y, B);
      dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
      long size = (C * (TILE_WIDTH) * (TILE_WIDTH) * sizeof(float));
      forward_kernel<<<gridDim, blockDim, size>>>(y.dptr_, x.dptr_, k.dptr_, B, M, C, H, W,
      K, W_grid);
```

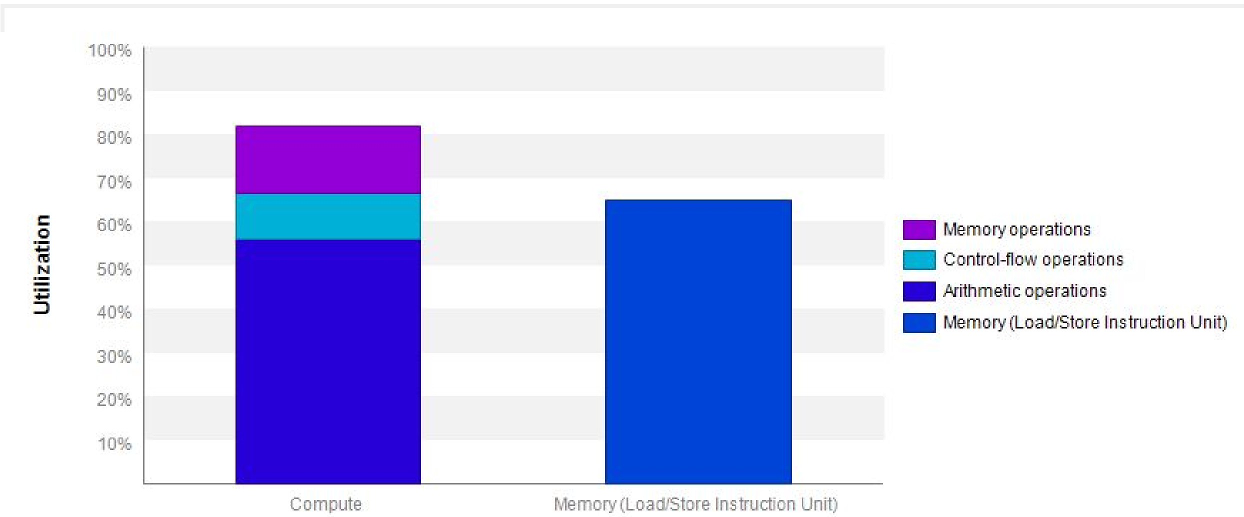
Performance Assessment:

Running time for python mp3.1py 100
New Inference
Op Time: 0.000742
Op Time: 0.001898
Correctness: 0.85 Model: ece408
44.24user 19.29system 1:01.59elapsed 103%CPU

Running time for python mp3.1py 1000
New Inference
Op Time: 0.007122
Op Time: 0.018504
Correctness: 0.827 Model: ece408
4.07user 2.44system 0:04.25elapsed 153%CPU

```
Running time for python mp3.1py 10000
New Inference
Op Time: 0.079162
Op Time: 0.186187
Correctness: 0.8171 Model: ece408
4.28user 2.74system 0:04.67elapsed 150%CPU
```

NVVP:



Strategy 2

Shared Memory

Shared Loads	734980515	3,783.903 GB/s	
Shared Stores	30700682	158.056 GB/s	
Shared Total	765681197	3,941.959 GB/s	Idle Low Medium High Max

Strategy 2

Shared Memory

Shared Loads	379625982	2,616.202 GB/s	
Shared Stores	10905444	75.155 GB/s	
Shared Total	390531426	2,691.357 GB/s	Idle Low Medium High Max

Strategy 3

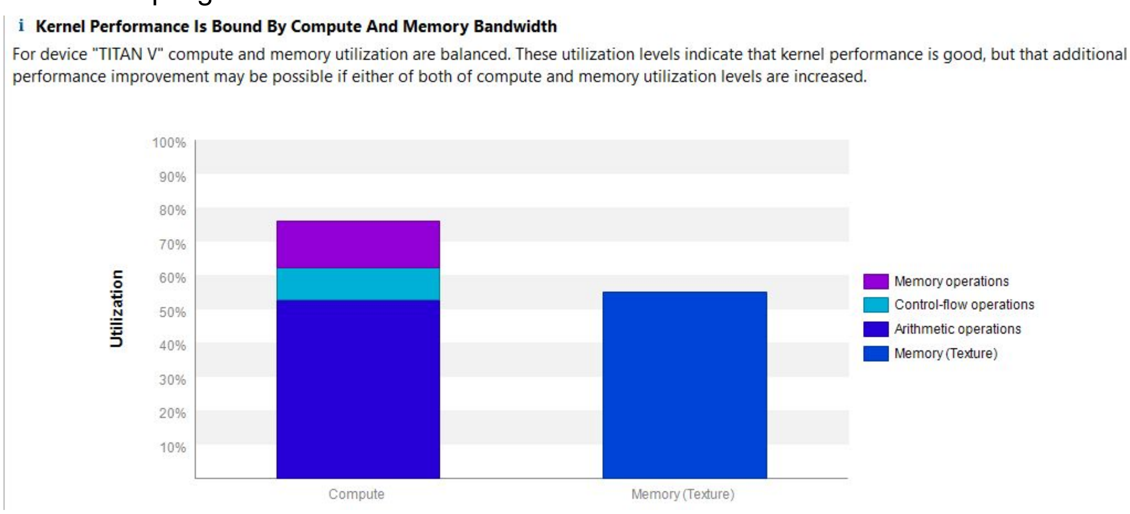
Using Strategy 2, while we were able to improve the memory utilization, we still observe that the number of global loads and stores are high. There are approximately 735M shared loads versus 815M global loads. Ideally, we would want to see a much higher shared memory access to improve performance further. Due to the way elements were loaded into shared memory, there was additional control divergence introduced, mainly due to the size of input images not being a multiple of 32. Strategy 3 of loading elements into the tiles was also explored, but the problem of control divergence was still present. However, due to smaller sizes for the second layer and larger block size, we see an improvement in the performance of the second layer using strategy 2. Neither of the strategies improved the overall performance significantly. To alleviate this, the matrix multiplication approach for convolution was explored as described in the previous two optimizations. The motivation is to exploit better control and memory divergence using matrix multiplication.

Milestone 3

Dataset	Correctness	Op Time 1 (s)	Op Time 2 (s)	User + System Time (s)
100	0.85	0.000592	0.001602	6.49
1000	0.827	0.005725	0.015483	6.60
10000	0.8171	0.056734	0.139802	6.61

Nvprof was used to profile the dataset with 100 images to get an overview of the kernels. The following properties were studied to determine performance limiting factors:

1. Global memory efficiency
2. Occupancy
3. Thread divergence
4. PC sampling

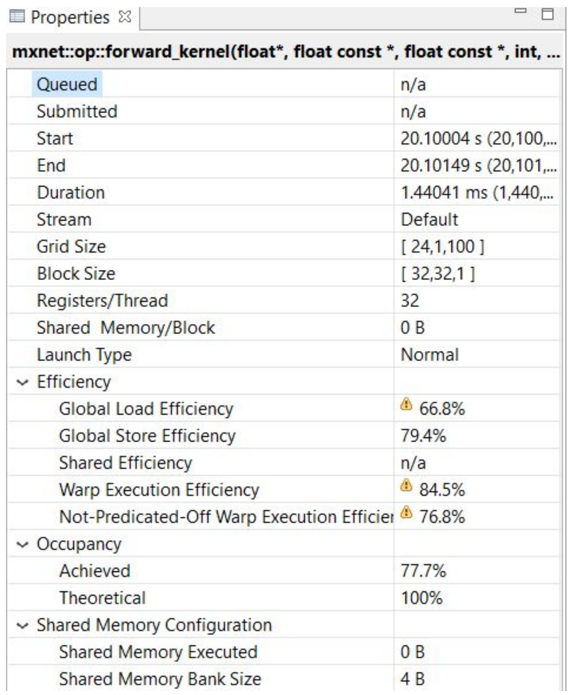


Since the current kernel is a naive GPU implementation of convolution, the aim of this exercise was to understand the different properties that can be observed and correspondingly performance optimizations can be targeted.

The following observations were made for the second instance of the forward kernel:

1. Global load and store efficiencies were 66.8% and 79.4% respectively. There is room for optimization in the way memory is accessed.

2. While the theoretical occupancy is 100%, only 77.7% occupancy was actually achieved, providing room for optimization here as well.
3. In this milestone, we haven't implemented shared memory optimization. Hence we observe the shared efficiency is n/a and the shared memory executed is 0B as shown in the figure. Hence there is room for improvement and increase in the performance using shared memory for optimizing the convolution layers.
4. We also observe that the duration of execution of kernel is 1.44041 ms as shown in the figure.

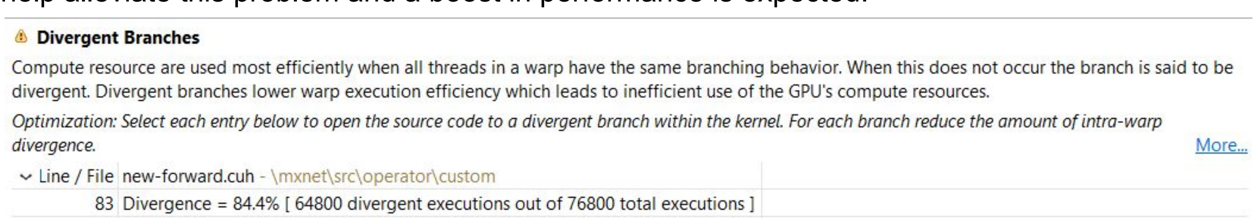


Properties

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

Queued	n/a
Submitted	n/a
Start	20.10004 s (20,100,...
End	20.10149 s (20,101,...
Duration	1.44041 ms (1,440,...
Stream	Default
Grid Size	[24,1,100]
Block Size	[32,32,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Global Load Efficiency	66.8%
Global Store Efficiency	79.4%
Shared Efficiency	n/a
Warp Execution Efficiency	84.5%
Not-Predicated-Off Warp Execution Efficier	76.8%
Occupancy	
Achieved	77.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

5. The current implementation shows 84.8% divergence. Defining better thread blocks will help alleviate this problem and a boost in performance is expected.



Divergent Branches

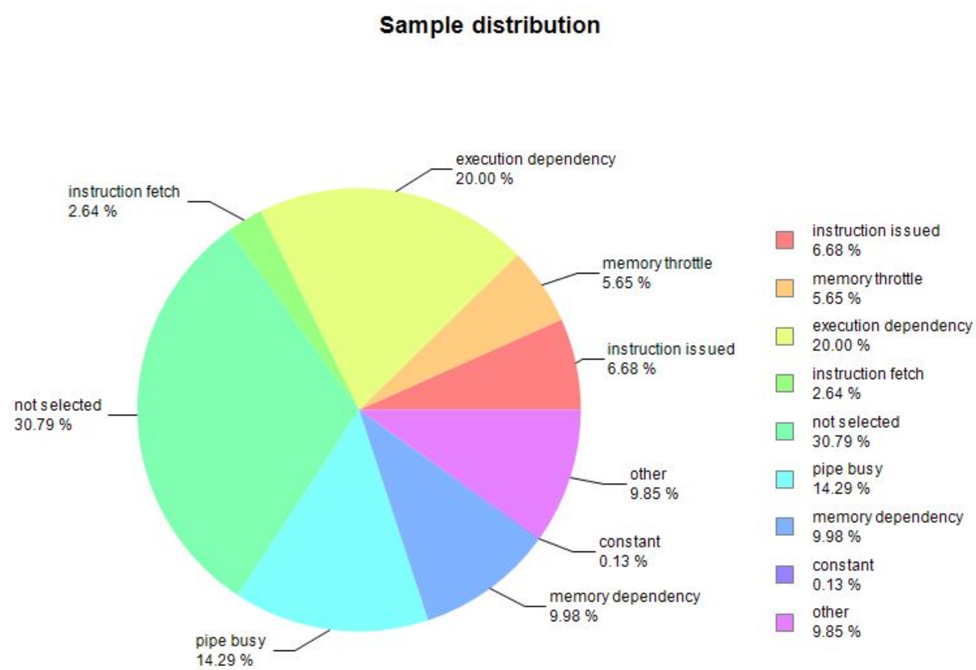
Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

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
83	Divergence = 84.4% [64800 divergent executions out of 76800 total executions]

6. PC sampling was studied to understand the distribution of time spent by the kernel in different operations like memory operations, execution operations, and so on. Since it is well distributed, the kernel is performing equally good (or bad) in each of the operations.



Milestone 2

Program execution time:

`133.47user 4.61system 2:07.56elapsed`

`Program run time: 138.08 s`

Op Times:

`Op Time: 21.291906 s`

`Op Time: 101.988109 s`

Milestone 1

1. Kernels that collectively consume more than 90% of the program time

   `36.82%  [CUDA memcpy HtoD]`

   `22.74%  volta_scudnn_128x32_relu_interior_nn_v1`

```
20.76%  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)

7.39%  volta_sgemm_128x128_tn

7.25%  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*)
32%  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)

0.52% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto,
int=8, int=1024, shadow::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)

0.07% 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)

0.06% 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)

0.03% volta_sgemm_32x32_sliced1x4_tn
```

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

```
38.66%  cudaStreamCreateWithFlags
34.05%  cudaMemGetInfo
21.64%  cudaFree
1.74%  cudaFuncSetAttribute
1.33%  cudaMalloc
1.10%  cudaMemcpy2DAsync
0.85%  cudaStreamSynchronize
0.28%  cudaEventCreateWithFlags
0.18%  cudaEventCreate
0.07%  cudaGetDeviceProperties
```

3. Difference between kernel and API calls

   Kernels are automatically loaded during initialization and stay loaded for as long as the program runs whereas with the API calls it is possible to only load modules that are currently needed or load them dynamically during runtime as well.

   Kernel functions are defined by the user to run computation on a GPU device called by the host using the __global__ declaration whereas API calls are defined by the CUDA library to perform predefined functions.

   Kernel is executed N time parallelly where N is the total number of threads whereas API calls are executed once.

4. Output of rai running MXNet on the CPU

```
EvalMetric: {'accuracy': 0.8177}
```

```
20.01user 4.13system 0:13.60elapsed 177%CPU (0avgtext+0avgdata
5954888maxresident)k
0inputs+2856out
puts (0major+1585429minor)pagefaults 0swaps
```

5. CPU program run time

```
20.01user 4.13system 0:13.60elapsed
Program run time : 24.14 s
```

6. Output of rai running MXNet on the GPU

```
EvalMetric: {'accuracy': 0.8177}
4.00user 2.59system 0:04.56elapsed 144%CPU (0avgtext+0avgdata
2841584maxresident)k
8inputs+1712outputs (0major+704309minor)pagefaults 0swaps
```

7. GPU program run time

```
4.00user 2.59system 0:04.56elapsed
Program run time: 6.59 s
```