# 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)

# Final Submission (52.148ms)

## Optimization 4: Kernel Fusion

1. Physical unrolling in shared memory without memory coalescing

   We initially implemented unrolling with shared matrix multiplication with unrolled matrix for all images in global memory. 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. We had to loop in batches of 1000 images to make it work.

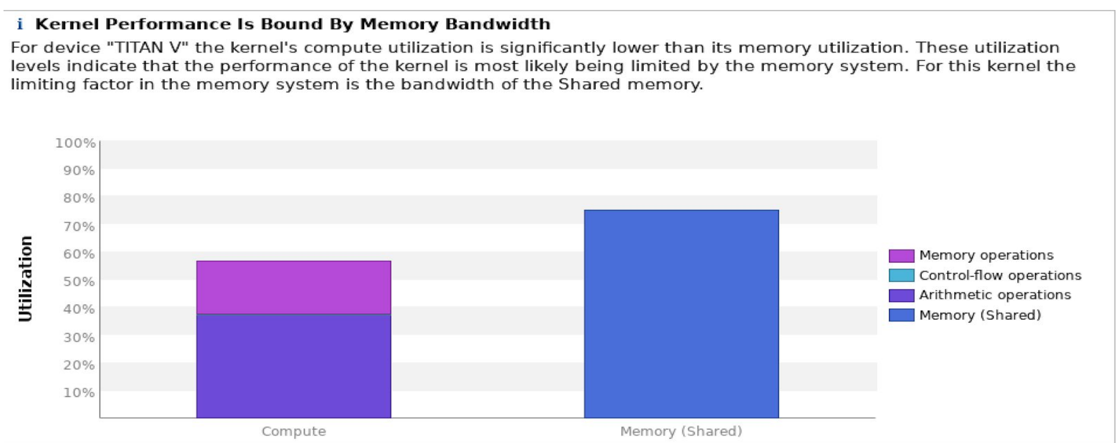
   This motivated us to load the elements of input matrix into shared memory and then physically unroll it in the shared memory and thereby overcome the issue of running out of global memory bandwidth due to unrolling. This required us to merge the kernel for unrolling and matrix multiplication.

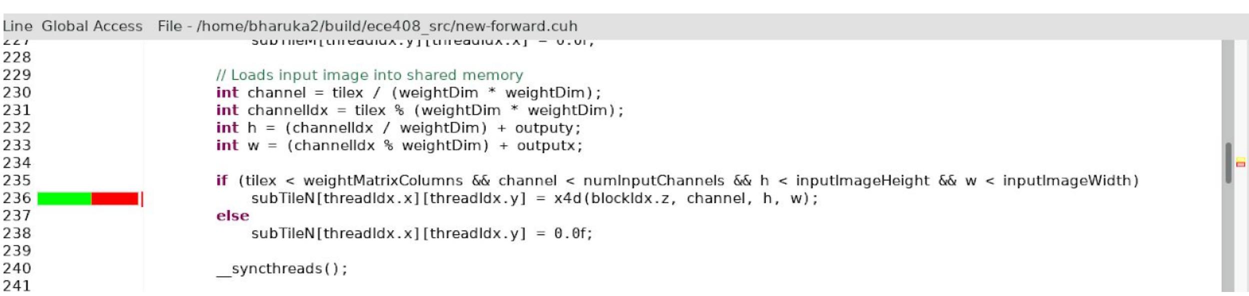
```
__global__ void forward_kernel_logical_unroll(const float* __restrict__ x, const
float* __restrict__ w, float* __restrict__ y, const int numImages, const int
numInputChannels, const int inputImageHeight, const int inputImageWidth, const int
weightDim, const int numOutputChannels, const int outputMatrixWidth, const int
outputImageWidth) {
   #define x4d(b,m,h,w) x[(b) * (numInputChannels * inputImageHeight *
inputImageWidth) + (m) * (inputImageHeight * inputImageWidth) + (h) *
(inputImageWidth) + w]

   float value = 0;
```

```
   const unsigned int row = blockDim.y * blockIdx.y + threadIdx.y;
   const unsigned int column = blockDim.x * blockIdx.x + threadIdx.x;
   __shared__ float subTileM[L2_TILE_WIDTH][L2_TILE_WIDTH];
   __shared__ float subTileN[L2_TILE_WIDTH][L2_TILE_WIDTH];

   const unsigned int weightMatrixColumns = numInputChannels * weightDim *
weightDim;
   const unsigned int columnStartIndex = blockDim.x * blockIdx.x + threadIdx.y;
   const unsigned int outputy = columnStartIndex / outputImageWidth;
   const unsigned int outputx = columnStartIndex % outputImageWidth;

   for (unsigned int i = 0; i < ceil(weightMatrixColumns, L2_TILE_WIDTH); i++) {
       // Loads weights into shared memory
       int tilex = i * L2_TILE_WIDTH + threadIdx.x;
       if (tilex < weightMatrixColumns && row < numOutputChannels)
           subTileM[threadIdx.y][threadIdx.x] = w[(row * weightMatrixColumns) +
tilex];
       else
           subTileM[threadIdx.y][threadIdx.x] = 0.0f;

       // Loads input image into shared memory
       int channel = tilex / (weightDim * weightDim);
       int channelIdx = tilex % (weightDim * weightDim);
       int h = (channelIdx / weightDim) + outputy;
       int w = (channelIdx % weightDim) + outputx;

       if (tilex < weightMatrixColumns && channel < numInputChannels && h <
inputImageHeight && w < inputImageWidth)
           subTileN[threadIdx.x][threadIdx.y] = x4d(blockIdx.z, channel, h, w);
       else
           subTileN[threadIdx.x][threadIdx.y] = 0.0f;
       __syncthreads();

       for (unsigned int j = 0; j < L2_TILE_WIDTH; j++) {
           value += subTileM[threadIdx.y][j] * subTileN[j][threadIdx.x];
       }
       __syncthreads();
   }
```

```
    if (row < numOutputChannels && column < outputMatrixWidth) {
        y[(numOutputChannels * outputMatrixWidth * blockIdx.z) + (outputMatrixWidth * row) + column] = value;
        //y[(numOutputChannels * outputMatrixWidth * blockIdx.z) + (outputMatrixWidth * row) + column] = __half2float(value);
    }
}
```

```
Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.039939
Op Time: 0.109172
Correctness: 0.8171 Model: ece408
4.29user 2.80system 0:04.55elapsed 155%CPU

Total program execution: 7.09 seconds
Total kernel execution time: 0.149111 seconds
```



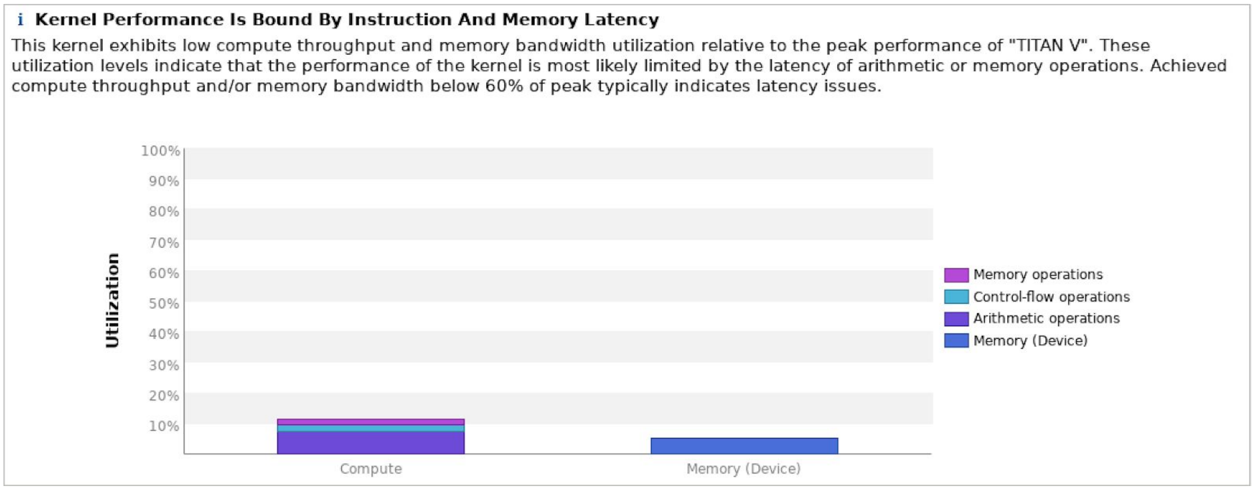



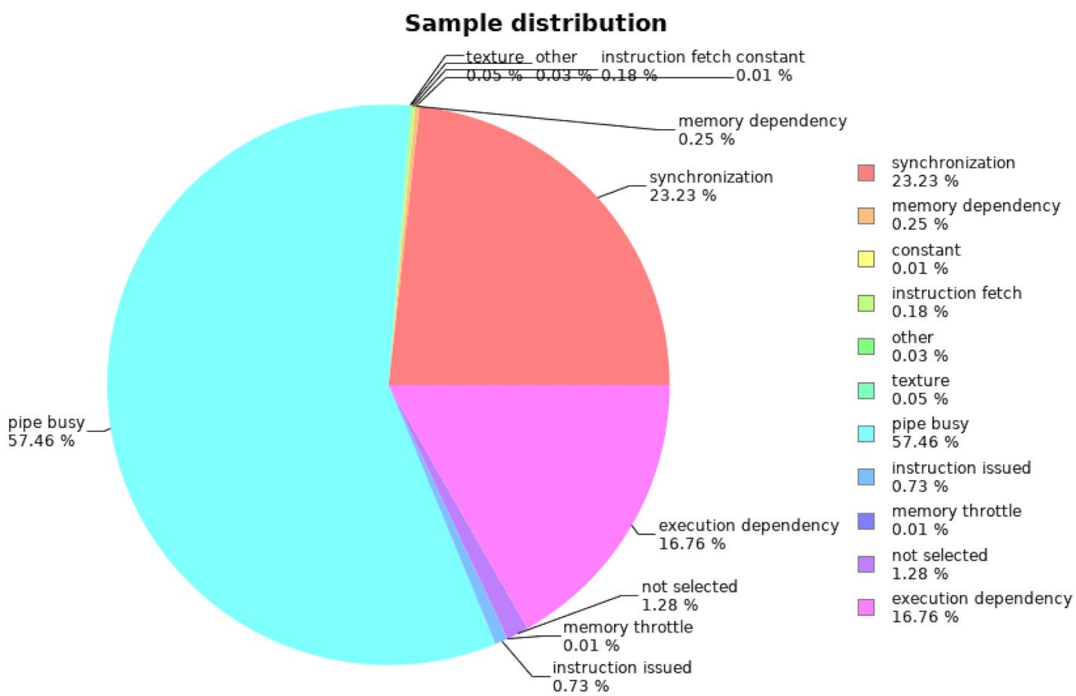
As seen in the utilization graph, we see that we are bound by global memory bandwidth. The reason is accessing the input matrix in a non-coalesced manner. To overcome global memory bandwidth, we implemented physical unrolling in shared memory with memory coalescing as shown below.

2. Physical unrolling in shared memory with memory coalescing

```
...
   for (unsigned int channelNum = 0; channelNum < numInputChannels; channelNum++) {
       if (threadIndex < weightDim*inputImageWidth) {
           float load_val = x[((blockIdx.z) * numInputChannels * inputImageHeight *
inputImageWidth) + (channelNum * inputImageHeight * inputImageWidth) + (
(inputImageRow + blockIdx.x) * inputImageWidth) + inputImageCol];

           int outputRow = inputImageRow * weightDim;
           int outputCol = inputImageCol;
           for (unsigned int i = 0; i < weightDim; i++) {
               if (outputCol >= 0 && outputCol < outputImageWidth)
                   subTileN[outputRow][outputCol] = load_val;
               outputCol -= 1;
               outputRow += 1;
           }
       }
...
```

```
Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.026953
Op Time: 0.083349
Correctness: 0.8171 Model: ece408
4.21user 2.79system 0:04.76elapsed 147%CPU

Total program execution time: 7 seconds
Total kernel execution time: 0.110302 seconds
```




As shown in the figure, we reduced our global memory bandwidth compared to the previous optimization. The difference from the previous optimization is to load the input matrix elements into shared memory in a coalesced manner.

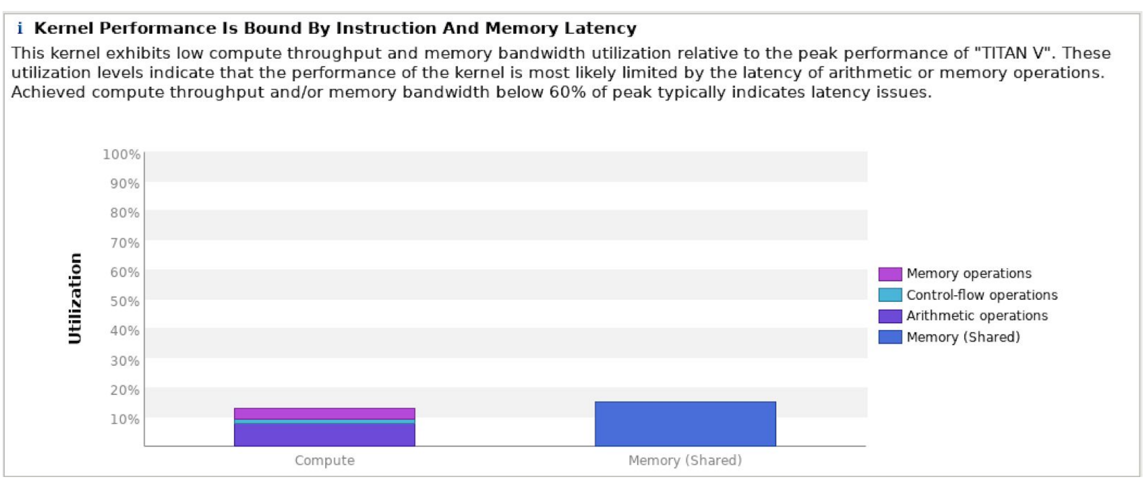
The other benefit we observed in nvprof was the compute bandwidth also reduced compared to our previous optimization. The reason behind this is that the access time to constant memory was keeping the pipeline busy and we were bound by latency

## Optimization 5: Thread Coarsening

We tried thread coarsening as one of the optimization techniques. The results are shown below

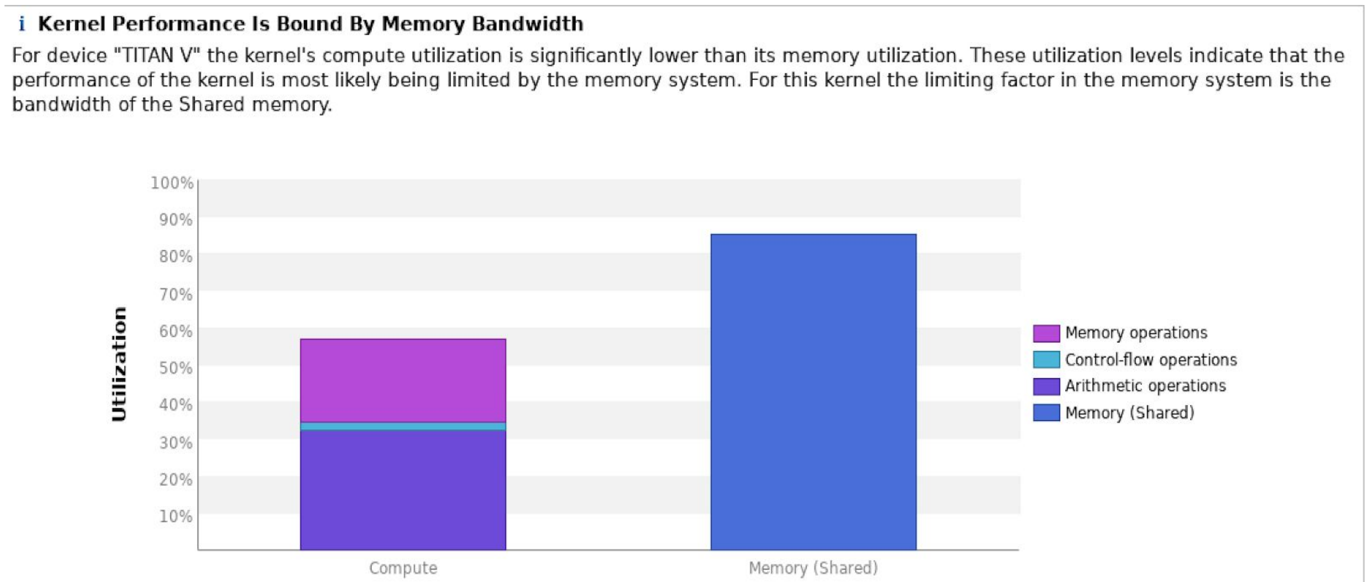
```
...
          float load_val = x[((4 * blockIdx.z) * numInputChannels *
inputImageHeight * inputImageWidth) + (channelNum * inputImageHeight *
inputImageWidth) + ( (inputImageRow + blockIdx.x) * inputImageWidth) +
inputImageCol];
          float load_val_O = x[((4 * blockIdx.z + 1 )* numInputChannels *
inputImageHeight * inputImageWidth) + (channelNum * inputImageHeight *
inputImageWidth) + ( (inputImageRow + blockIdx.x) * inputImageWidth) +
inputImageCol];
          float load_val_P = x[((4 * blockIdx.z + 2 )* numInputChannels *
inputImageHeight * inputImageWidth) + (channelNum * inputImageHeight *
inputImageWidth) + ( (inputImageRow + blockIdx.x) * inputImageWidth) +
inputImageCol];
          float load_val_Q = x[((4 * blockIdx.z + 3 )* numInputChannels *
inputImageHeight * inputImageWidth) + (channelNum * inputImageHeight *
inputImageWidth) + ( (inputImageRow + blockIdx.x) * inputImageWidth) +
inputImageCol];
...
```

We implemented thread coarsening by computing 3 images together in layer 1 and 4 images together in layer 2. We used constant memory to load the weight matrix elements and shared memory to load the input matrix elements. The performance of this optimization technique shown in the figure below.



We observe from the above figure that we are latency bound. However we are not bounded by memory bandwidth anymore. Since we are latency bound, we don't get a good performance.

To improve the above optimization, we used shared memory to load the elements from weight matrix and input matrix. The performance results are shown below.

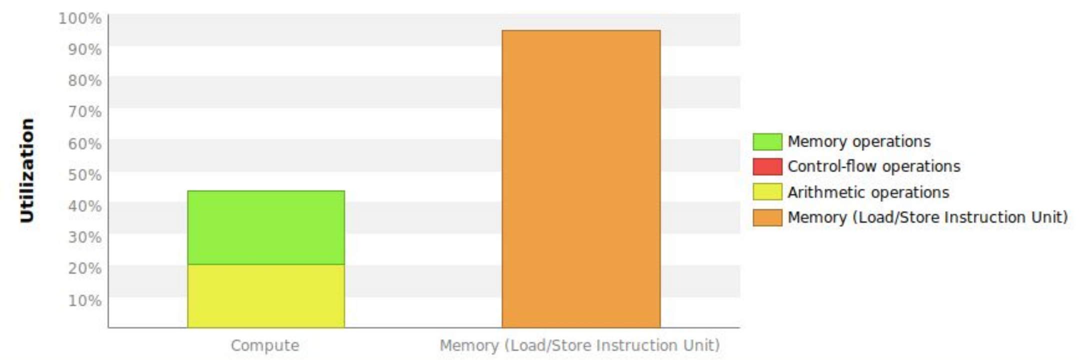



As seen in the above figure, we observe that we are bound by memory bandwidth. The nvprof tells that the shared memory stores while accessing shared memory for writing the weights is unaligned. Shared memory bank conflicts could be a reason for this behavior.

```
Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.019331
Op Time: 0.046854
Correctness: 0.8171 Model: ece408
4.30user 2.45system 0:04.58elapsed 147%CPU

Total program execution time: 6.75 seconds
Total kernel execution time: 0.066185 seconds
```

## Optimization 6: Separate Kernels for Layers

In order to improve the performance of Layer 1, we implemented shared memory convolution (without matrix multiplication) and used constant memory to load the elements of weight matrix. For the second layer, we implemented unrolling + shared matrix multiplication. This improved the overall performance since the performance of layer 1 improved drastically.
The performance is as shown below.




As seen in the above diagram, we don't see control flow operations due to implementation of unrolling. However, we are bound by memory bandwidth due to shared memory bank conflicts. Compute bound reduced due to shared memory convolution implementation in Layer1.

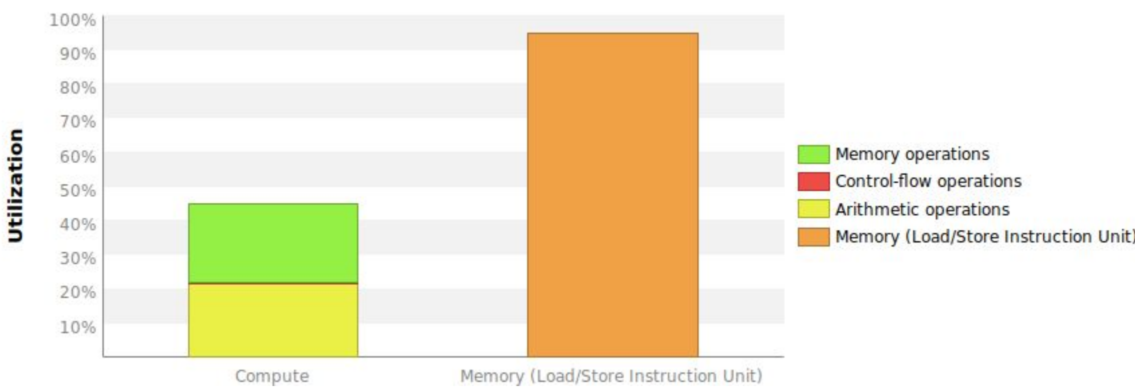
```
✱ Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.008122
Op Time: 0.044911
Correctness: 0.8171 Model: ece408
4.17user 2.53system 0:04.57elapsed 146%CPU (0avgtext+0avgdata
2840500maxresident)k

Total program execution time: 4.44906406 seconds
Total kernel execution time: 0.053033 seconds (53 milli seconds)
```

## Optimization 7: Exploiting parameters, tuning with restrict, loop unrolling and other software optimizations

Some of the compiler optimization techniques we implemented are:

1. Loop unrolling - #pragma unroll
2. Use of restrict keyword - Tells the compiler that ptr is the only way to access the object pointed by it and compiler doesn't need to add any additional checks
3. Use of register variable - The keyword register hints to compiler that a given variable can be put in a registers
4. Reducing redundant integer computations such as number of divide and modulo operations
5. Use of fmaf (fused multiply - add operation) - This performs multiplication and addition operation in a single cycle
6. We also tried implementing half precision floating point for multiplying the weight with the element and using single precision floating point for adding the resultant multiply and accumulate to the output. The performance of the kernel degraded. The primary reason for this degradation was the overhead of casting to and from single precision to half precision.

On removing restrict keyword and unrolling from our code, we get the following performance:



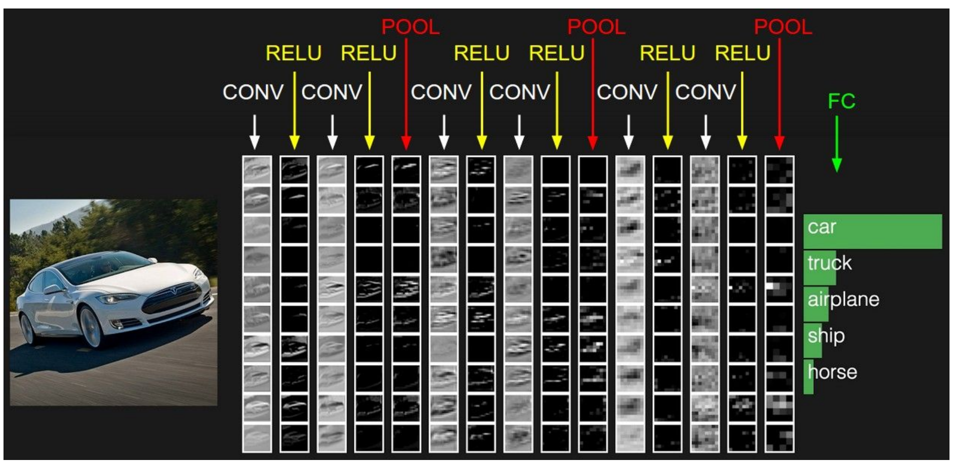

As shown in the above picture, we see an increase in the control-flow operations compared to our previous optimization techniques due to removal of unrolling. As explained before, the memory bandwidth is high due to shared memory bank conflicts.

```
Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.008090
Op Time: 0.045427
Total kernel execution time: 0.053517 seconds
```

# Future work

## Compressed Convolution

Each layer in a convolutional neural network usually is a set of multiple operations, such as convolution, ReLU (previously, sigmoids were commonly used), and subsampling (like a maxpool). Due to property of ReLU and subsampling, the resultant layer has a relatively high occurrence of three values - 0’s, 1’s and -1’s. This can be coarsely visualized from the image below. The output of the first ReLU layer will have a high occurrence of the three values and will serve as input to the next layer. If we can compress this image, the number of calculations can reduce. From the first five images in the dataset of the project, an average of ~30% elements were 0’s, 1’s and -1’s. If there are K x K elements with one of the three values, we can effectively reduce 49 multiply-and-adds by 1 move, given we have computed the value of a kernel on such a block previously. Sparse matrix techniques could help to exploit this property and potentially boost performance of the convolution layer.



# 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:

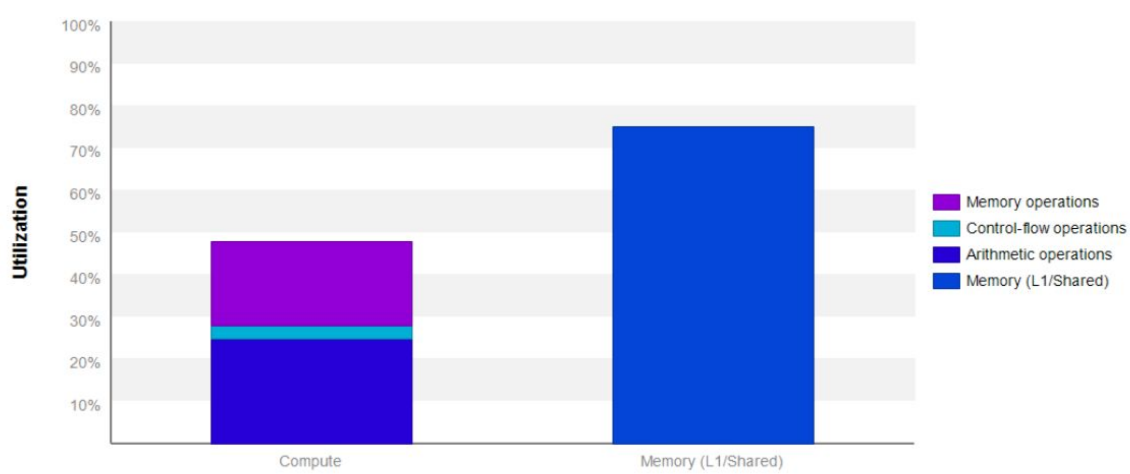
```
...
dim3 gridMatrix((TILE_WIDTH+matrixWidth-1)/TILE_WIDTH, (TILE_WIDTH+M-1)/TILE_WIDTH, B);
dim3 blockMatrix(TILE_WIDTH, TILE_WIDTH, 1);

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
```

```
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
```

```
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:

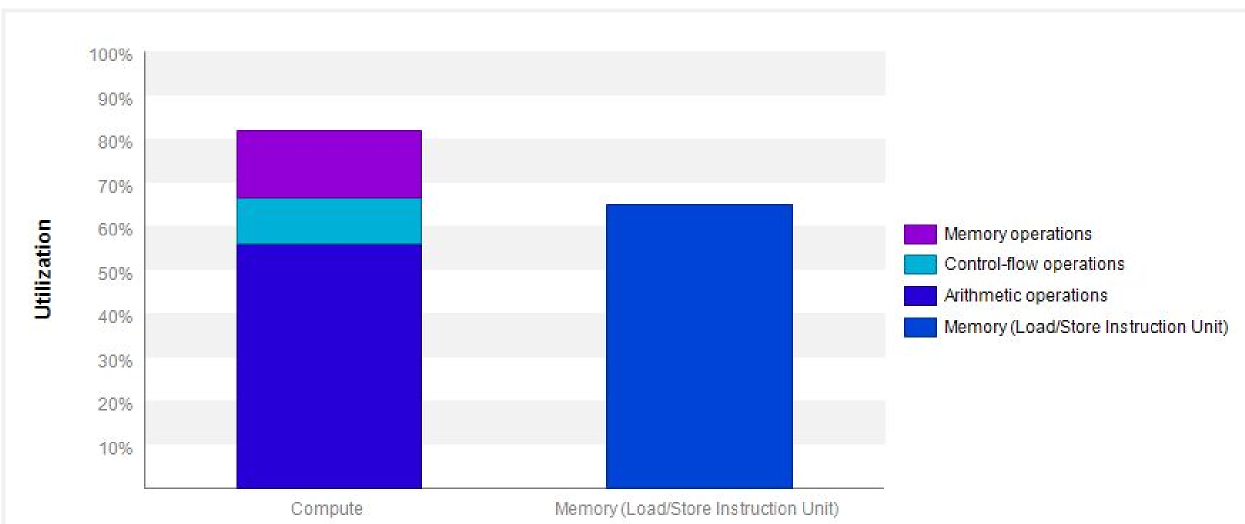
Running time for python mp3.1py 100

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**

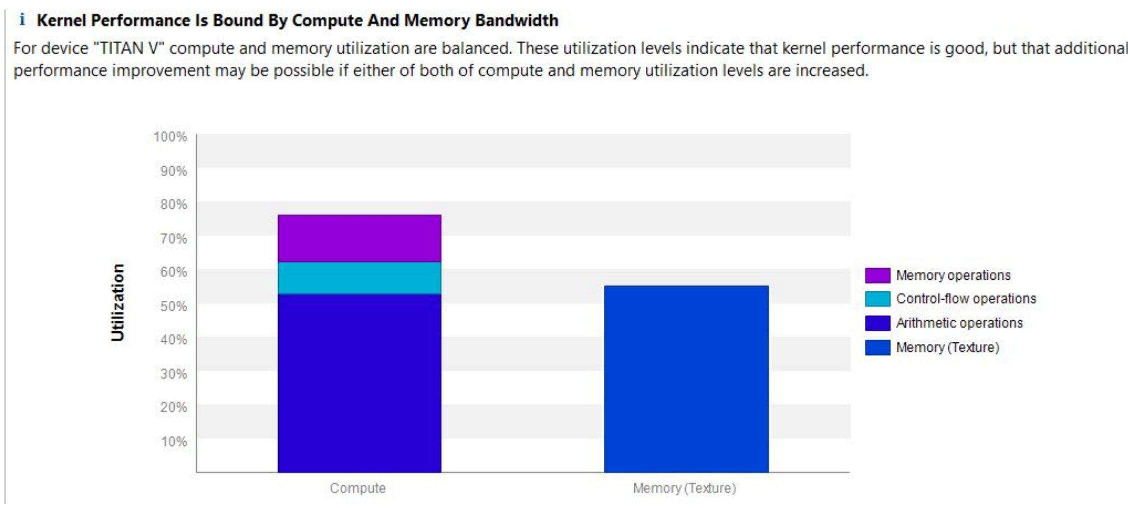
Using Strategy 2, while we were able to improve the memory utilization, we still observe that the number of global loads and stores are high. There are approximately 735M shared loads versus 815M global loads. Ideally, we would want to see a much higher shared memory access to improve performance further. 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



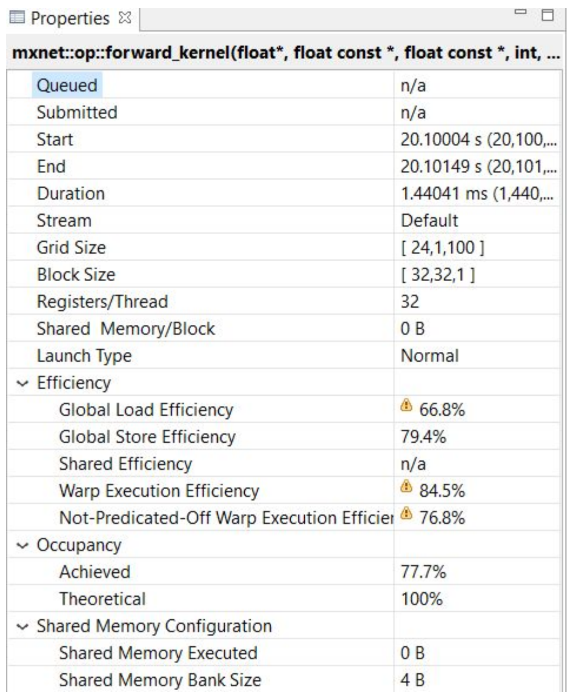

Since the current kernel is a naive GPU implementation of convolution, the aim of this exercise was to understand the different properties that can be observed and correspondingly performance optimizations can be targeted.

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.

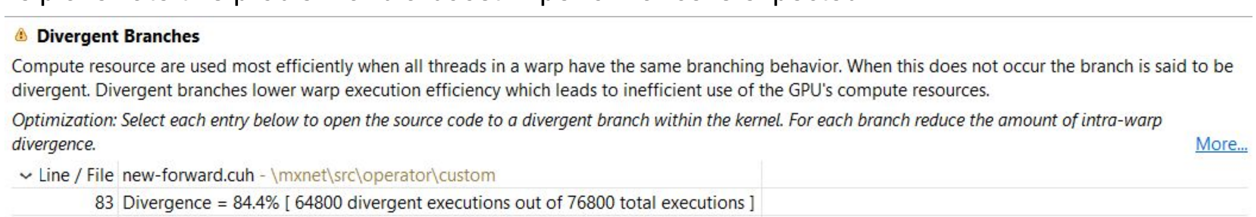


Properties

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

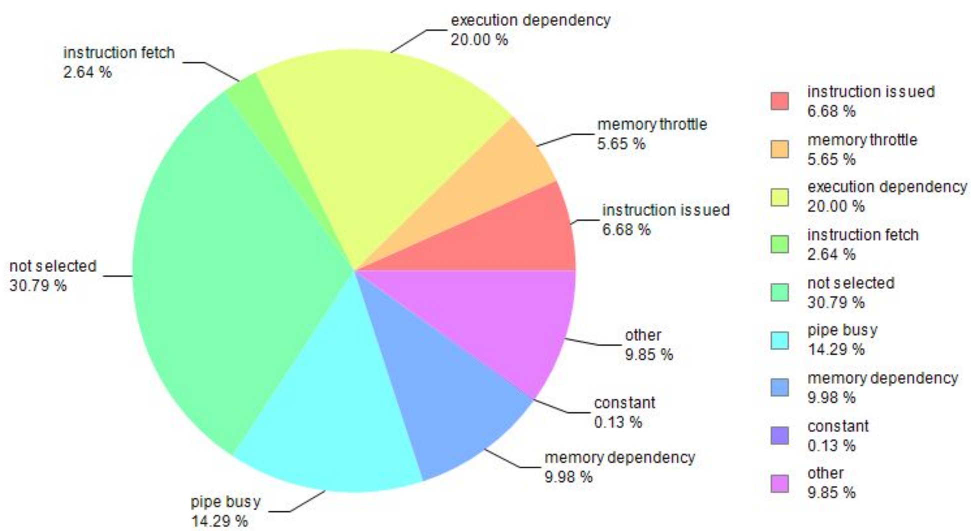
| | |
|---|---|
| Queued | n/a |
| Submitted | n/a |
| Start | 20.10004 s (20,100,... |
| End | 20.10149 s (20,101,... |
| Duration | 1.44041 ms (1,440,... |
| Stream | Default |
| Grid Size | [ 24,1,100 ] |
| Block Size | [ 32,32,1 ] |
| Registers/Thread | 32 |
| Shared Memory/Block | 0 B |
| Launch 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.




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.

## Sample distribution



# 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
```