ECE 408/CS 483 Final Project

Team: elegant_and_easygoing_boys
Team members:
Licheng Luo (NetID: ll6) (UIN: 676011317)
Zhengqi Fang (NetID: zf4) (UIN: 678965516)
Ruian Pan (NetID: ruianp2) (UIN:659336387)
Affiliation: on campus

Milestone 2

All kernels that collectively consume more than 90% of the program time

volta_scudnn_128x64_relu_interior_nn_v1
volta_gcgemm_64x32_nt
fft2d_c2r_32x32
volta_sgemm_128x128_tn
op_generic_tensor_kernel
fft2d_r2c_32x32
cudnn::detail::pooling_fw_4d_kernel

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

cudaStreamCreateWithFlags
cudaMemGetInfo
cudaFree

Explanation of the difference between kernels and API calls

Kernels are the codes that run on GPU and do the parallel computations.
API calls are the calls to the CUDA's APIs, which are defined by CUDA(NVIDIA). They are usually used to do the initializations such as memory allocations and transfer.

Output of RAI running MXNet on the CPU (m1.1)

EvalMetric: {'accuracy': 0.8154}

Run time

User	20.89

System	7.39
Elapsed	0:10.28

Output of RAI running MXNet on the GPU (m1.2)

EvalMetric: {'accuracy': 0.8154}
Run time

User	5.10
System	2.69
Elapsed	0:05.01

CPU Implementation

Correctness: 0.7653 Model: ece408
Run time (m2.1)

User	90.33
System	10.19
Elapsed	1:19.22

Op Times

Op Time: 13.540072
Op Time: 60.894102

Milestone 3

Result

	Data size 100	Data size 1000	Data size 10000
Correctness	0.76	0.767	0.7653
Timing			

	User	System	Elapsed	User	System	Elapsed	User	System	Elapsed
	5.03	2.88	0:04.53	6.55	3.32	0:06.55	25.01	9.50	0:31.01

Nvprof and NVVP performance analysis

NVVP trace (data size: 100)

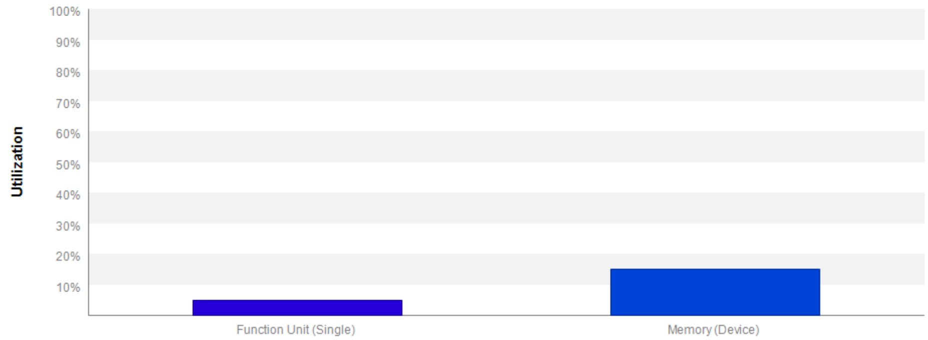
Note that 99.6% of the Compute is the forward_kernel.

Analysis

We did a detailed analysis on kernel 1, and NVVP showed that the kernel performance is bound by instruction and memory latency, as the screenshot shown below:

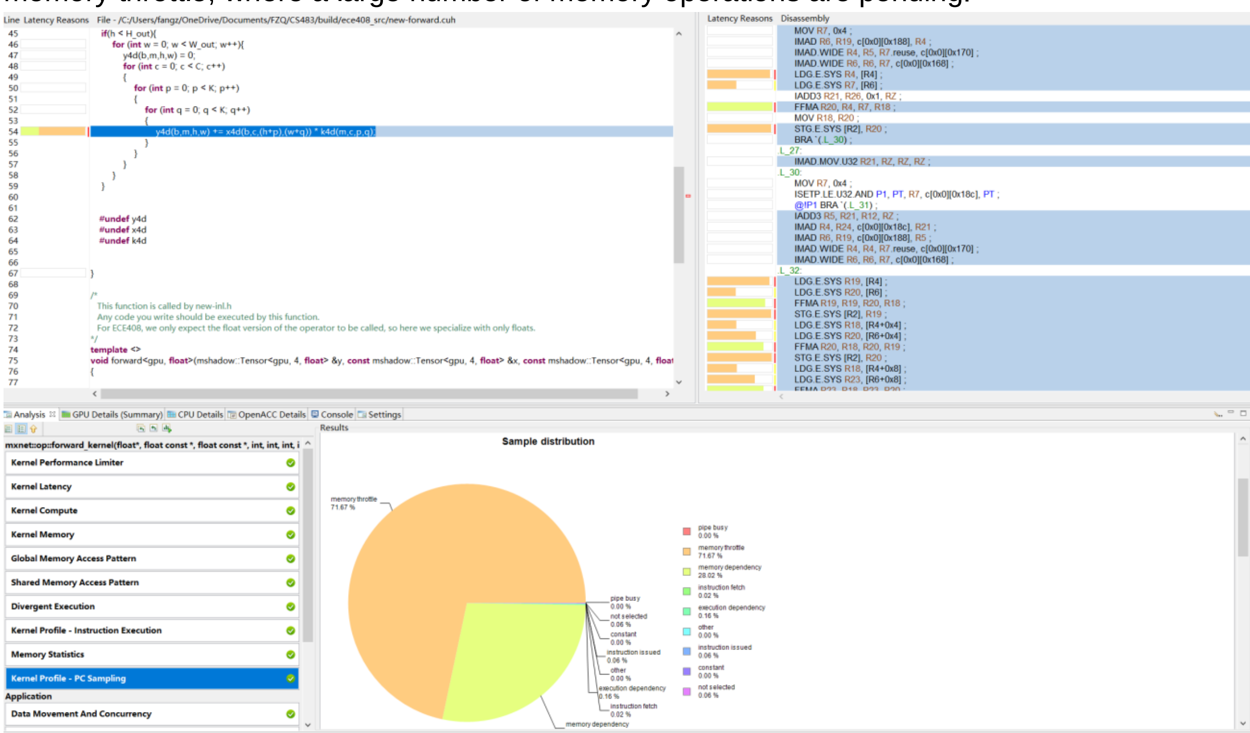
Kernel Performance Is Bound By Instruction And Memory Latency

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.



We believe this is because we don’t use any shared memory to do the tiling, which makes the memory bandwidth really small. In the future, we are going to try to utilize the shared memory to load the input in order to improve the bandwidth.

We also ran the PC sampling analysis. We can see from below that the main bottleneck is the memory throttle, where a large number of memory operations are pending.



Milestone 4

The three optimizations we use are **Shared Memory convolution, Weight matrix (kernel values) in constant memory** and **Tuning with restrict and loop unrolling.**

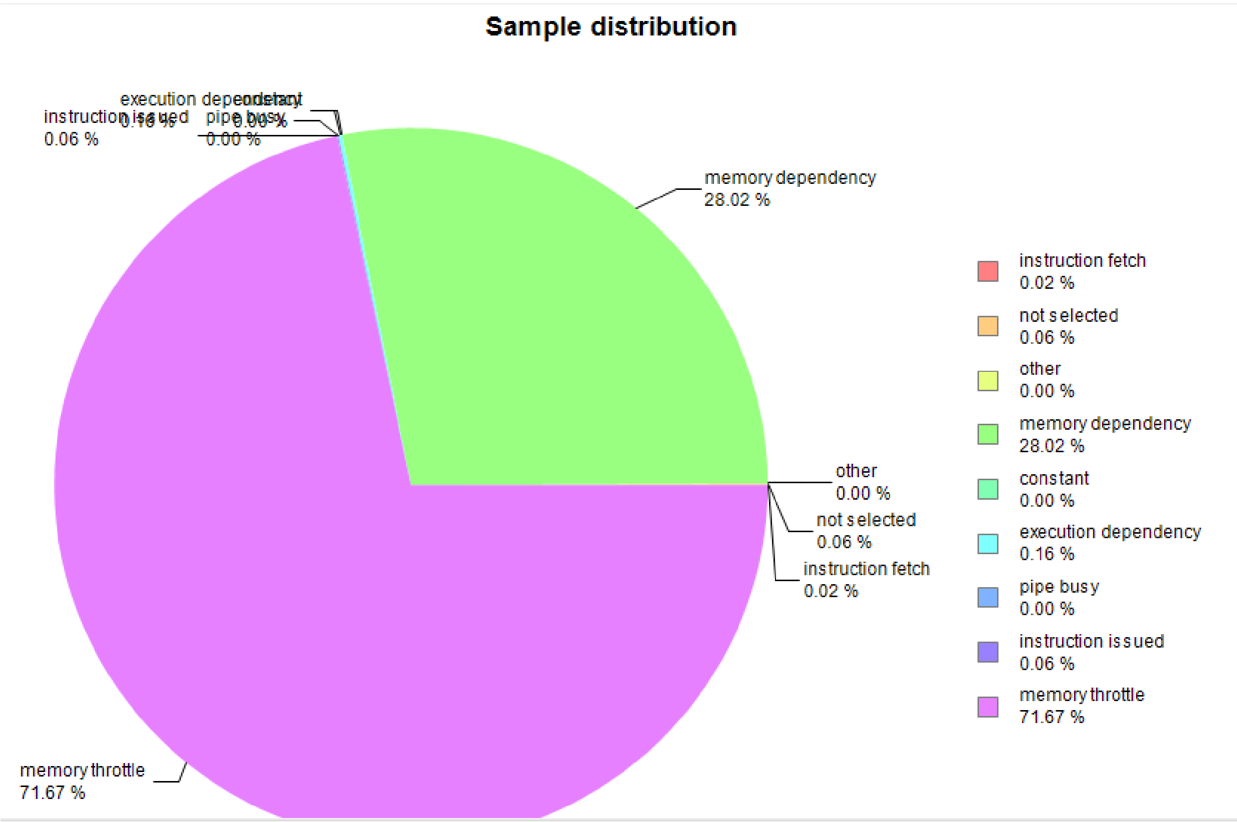
Note that for this section, we are using data size of 100 for NVVP analysis.

Optimization 1: Shared Memory convolution

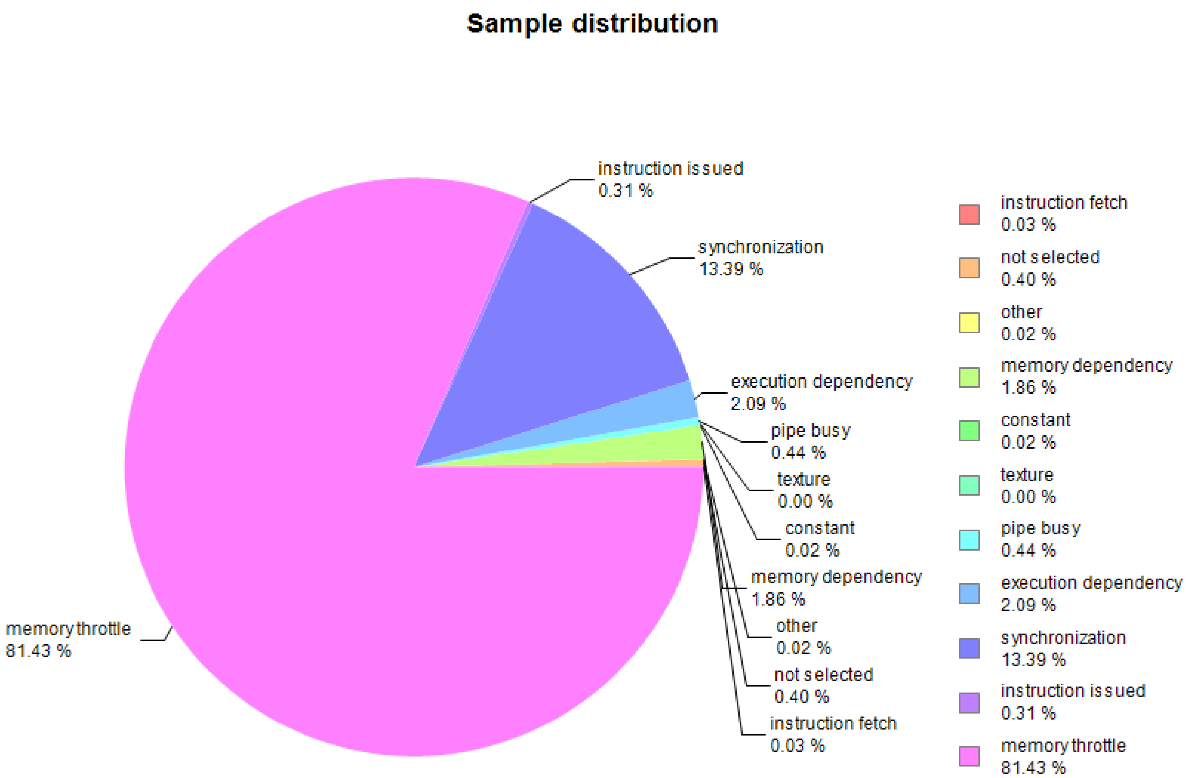
The shared memory convolution can greatly reduce the global memory access of x array. By reusing the tiling region, and keeping a reduced number of global access, it improves the performance of our code. (See the red portion of the code for our usage).

NVVP analysis

Before:



After:



Compare the sample distribution before and after the optimization, we can see that shared memory greatly reduced the latencies caused by memory dependency. Although synchronization becomes significant now, the benefit it brings compensates the cost.

(Zhengqi and Licheng worked on this optimization.)

Optimization 2: Tuning with restrict and loop unrolling

Restrict keyword in the function definition would alleviate the aliasing problem that exists in C-type languages. It allows reordering and doing common sub-expression elimination.

Loop unrolling can avoid the loop structure in the compiled code. Instead of using the loop structure to control the code in the compiled code, the loop unrolled code would be compiled to serialized code to reduce overhead and improve efficiency.
(See the purple portion of the code for our usage).

NVVP Analysis
Before:

Kernel Profile - Instruction Execution

The Kernel Profile - Instruction Execution shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

Cuda Functions :

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

Maximum instruction execution count in assembly: 816750

Average instruction execution count in assembly: 472359

After:

Kernel Profile - Instruction Execution

The Kernel Profile - Instruction Execution shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

Cuda Functions :

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

Maximum instruction execution count in assembly: 378000

Average instruction execution count in assembly: 195881

We can see that the instruction execution count in assembly is greatly reduced due to the loop unrolling, proving that this optimization successfully reduce the overhead of for loop computations.

(Zhengqi and Ruian worked on this optimization.)

Optimization 3: Weight matrix (kernel values) in constant memory

The constant memory takes less time to read with cache and since the filter is not changed during execution, it’s a perfect candidate to use constant memory to improve performance. (See the blue portion of the code for our usage).

The Total Running Time Profiling:

<table>
<tr><th>Data Size 1000</th><th>Milestone 3 (No optimization)</th><th>Milestone 4 (With all optimizations)</th></tr>
<tr><td>Profiling using usr/bin/time</td>
<td><table><tr><td>User</td><td>System</td><td>Elapsed</td></tr><tr><td>6.55</td><td>3.32</td><td>0:06.55</td></tr></table></td>
<td><table><tr><td>User</td><td>System</td><td>Elapsed</td></tr><tr><td>5.34</td><td>3.39</td><td>0:05.03</td></tr></table></td></tr>
</table>

<table>
<tr><th>Data Size 10000</th><th>Milestone 3 (No optimization)</th><th>Milestone 4 (With all optimizations)</th></tr>
<tr><td>Profiling using usr/bin/time</td>
<td><table><tr><td>User</td><td>System</td><td>Elapsed</td></tr><tr><td>25.01</td><td>9.50</td><td>0:31.01</td></tr></table></td>
<td><table><tr><td>User</td><td>System</td><td>Elapsed</td></tr><tr><td>9.40</td><td>4.16</td><td>0:10.08</td></tr></table></td></tr>
</table>

Comparing the two input size, it makes sense that as the data size increases the optimized version would have a better performance as the overhead becomes comparably smaller than other parts of computation.

(Zhengqi and Licheng worked on this optimization.)

Code

```
const int TILE_WIDTH = 8;

__constant__ float MASK[7200];

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

    int b1 = blockIdx.x;
    int b2 = blockIdx.y;
    int b3 = blockIdx.z;
    int t1 = threadIdx.x;
    int t2 = threadIdx.y;
    int t3 = threadIdx.z;
    int m = b1 * TILE_WIDTH + t1;
    int h = b2 * TILE_WIDTH + t2;
    int w = b3 * TILE_WIDTH + t3;

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

    #define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) *
(W_out) + i0]
    #define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
    #define K4d(i3, i2, i1, i0) MASK[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    __shared__ float subTile[TILE_WIDTH][TILE_WIDTH][TILE_WIDTH];

    int currM = blockIdx.x * blockDim.x;
    int currH = blockIdx.y * blockDim.y;
    int currW = blockIdx.z * blockDim.z;
    int nextM = (blockIdx.x + 1) * blockDim.x;
    int nextH = (blockIdx.y + 1) * blockDim.y;
    int nextW = (blockIdx.z + 1) * blockDim.z;

    int mhw = m * (H_out * W_out) + h * (W_out) + w;
    for (int b = 0; b < B; b++)
    {
        int bmhw = b * (M * H_out * W_out) + mhw;
        if (m < M && h < H && w < W)
```

```
            subTile[t1][t2][t3] = x4d(b, m, h, w);
        else
            subTile[t1][t2][t3] = 0;
        __syncthreads();
        if (m < M && h < H_out && w < W_out)
            y[bmhw] = 0;
        for (int c = 0; c < C; c++)
        {
            if (m < M && h < H_out && w < W_out)
            {
                if (c >= currM && c < nextM && (h + 0) >= currH && (h + 0) < nextH && (w + 0) >=
currW && (w + 0) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 0][t3 + 0] * K4d(m, c, 0, 0);
                else
                    y[bmhw] += x4d(b, c, (h + 0), (w + 0)) * K4d(m, c, 0, 0);

                if (c >= currM && c < nextM && (h + 0) >= currH && (h + 0) < nextH && (w + 1) >=
currW && (w + 1) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 0][t3 + 1] * K4d(m, c, 0, 1);
                else
                    y[bmhw] += x4d(b, c, (h + 0), (w + 1)) * K4d(m, c, 0, 1);

                if (c >= currM && c < nextM && (h + 0) >= currH && (h + 0) < nextH && (w + 2) >=
currW && (w + 2) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 0][t3 + 2] * K4d(m, c, 0, 2);
                else
                    y[bmhw] += x4d(b, c, (h + 0), (w + 2)) * K4d(m, c, 0, 2);

                if (c >= currM && c < nextM && (h + 0) >= currH && (h + 0) < nextH && (w + 3) >=
currW && (w + 3) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 0][t3 + 3] * K4d(m, c, 0, 3);
                else
                    y[bmhw] += x4d(b, c, (h + 0), (w + 3)) * K4d(m, c, 0, 3);

                if (c >= currM && c < nextM && (h + 0) >= currH && (h + 0) < nextH && (w + 4) >=
currW && (w + 4) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 0][t3 + 4] * K4d(m, c, 0, 4);
                else
                    y[bmhw] += x4d(b, c, (h + 0), (w + 4)) * K4d(m, c, 0, 4);

                if (c >= currM && c < nextM && (h + 1) >= currH && (h + 1) < nextH && (w + 0) >=
currW && (w + 0) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 1][t3 + 0] * K4d(m, c, 1, 0);
                else
                    y[bmhw] += x4d(b, c, (h + 1), (w + 0)) * K4d(m, c, 1, 0);

                if (c >= currM && c < nextM && (h + 1) >= currH && (h + 1) < nextH && (w + 1) >=
currW && (w + 1) < nextW)
                    y[bmhw] += subTile[c - currM][t2 + 1][t3 + 1] * K4d(m, c, 1, 1);
                else
```

```
            y[bmhw] += x4d(b, c, (h + 1), (w + 1)) * K4d(m, c, 1, 1);

        if (c >= currM && c < nextM && (h + 1) >= currH && (h + 1) < nextH && (w + 2) >=
currW && (w + 2) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 1][t3 + 2] * K4d(m, c, 1, 2);
        else
            y[bmhw] += x4d(b, c, (h + 1), (w + 2)) * K4d(m, c, 1, 2);

        if (c >= currM && c < nextM && (h + 1) >= currH && (h + 1) < nextH && (w + 3) >=
currW && (w + 3) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 1][t3 + 3] * K4d(m, c, 1, 3);
        else
            y[bmhw] += x4d(b, c, (h + 1), (w + 3)) * K4d(m, c, 1, 3);

        if (c >= currM && c < nextM && (h + 1) >= currH && (h + 1) < nextH && (w + 4) >=
currW && (w + 4) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 1][t3 + 4] * K4d(m, c, 1, 4);
        else
            y[bmhw] += x4d(b, c, (h + 1), (w + 4)) * K4d(m, c, 1, 4);

        if (c >= currM && c < nextM && (h + 2) >= currH && (h + 2) < nextH && (w + 0) >=
currW && (w + 0) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 2][t3 + 0] * K4d(m, c, 2, 0);
        else
            y[bmhw] += x4d(b, c, (h + 2), (w + 0)) * K4d(m, c, 2, 0);

        if (c >= currM && c < nextM && (h + 2) >= currH && (h + 2) < nextH && (w + 1) >=
currW && (w + 1) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 2][t3 + 1] * K4d(m, c, 2, 1);
        else
            y[bmhw] += x4d(b, c, (h + 2), (w + 1)) * K4d(m, c, 2, 1);

        if (c >= currM && c < nextM && (h + 2) >= currH && (h + 2) < nextH && (w + 2) >=
currW && (w + 2) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 2][t3 + 2] * K4d(m, c, 2, 2);
        else
            y[bmhw] += x4d(b, c, (h + 2), (w + 2)) * K4d(m, c, 2, 2);

        if (c >= currM && c < nextM && (h + 2) >= currH && (h + 2) < nextH && (w + 3) >=
currW && (w + 3) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 2][t3 + 3] * K4d(m, c, 2, 3);
        else
            y[bmhw] += x4d(b, c, (h + 2), (w + 3)) * K4d(m, c, 2, 3);

        if (c >= currM && c < nextM && (h + 2) >= currH && (h + 2) < nextH && (w + 4) >=
currW && (w + 4) < nextW)
            y[bmhw] += subTile[c - currM][t2 + 2][t3 + 4] * K4d(m, c, 2, 4);
        else
            y[bmhw] += x4d(b, c, (h + 2), (w + 4)) * K4d(m, c, 2, 4);
```

```
            if (c >= currM && c < nextM && (h + 3) >= currH && (h + 3) < nextH && (w + 0) >=
currW && (w + 0) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 3][t3 + 0] * K4d(m, c, 3, 0);
            else
                y[bmhw] += x4d(b, c, (h + 3), (w + 0)) * K4d(m, c, 3, 0);

            if (c >= currM && c < nextM && (h + 3) >= currH && (h + 3) < nextH && (w + 1) >=
currW && (w + 1) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 3][t3 + 1] * K4d(m, c, 3, 1);
            else
                y[bmhw] += x4d(b, c, (h + 3), (w + 1)) * K4d(m, c, 3, 1);

            if (c >= currM && c < nextM && (h + 3) >= currH && (h + 3) < nextH && (w + 2) >=
currW && (w + 2) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 3][t3 + 2] * K4d(m, c, 3, 2);
            else
                y[bmhw] += x4d(b, c, (h + 3), (w + 2)) * K4d(m, c, 3, 2);

            if (c >= currM && c < nextM && (h + 3) >= currH && (h + 3) < nextH && (w + 3) >=
currW && (w + 3) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 3][t3 + 3] * K4d(m, c, 3, 3);
            else
                y[bmhw] += x4d(b, c, (h + 3), (w + 3)) * K4d(m, c, 3, 3);

            if (c >= currM && c < nextM && (h + 3) >= currH && (h + 3) < nextH && (w + 4) >=
currW && (w + 4) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 3][t3 + 4] * K4d(m, c, 3, 4);
            else
                y[bmhw] += x4d(b, c, (h + 3), (w + 4)) * K4d(m, c, 3, 4);

            if (c >= currM && c < nextM && (h + 4) >= currH && (h + 4) < nextH && (w + 0) >=
currW && (w + 0) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 4][t3 + 0] * K4d(m, c, 4, 0);
            else
                y[bmhw] += x4d(b, c, (h + 4), (w + 0)) * K4d(m, c, 4, 0);

            if (c >= currM && c < nextM && (h + 4) >= currH && (h + 4) < nextH && (w + 1) >=
currW && (w + 1) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 4][t3 + 1] * K4d(m, c, 4, 1);
            else
                y[bmhw] += x4d(b, c, (h + 4), (w + 1)) * K4d(m, c, 4, 1);

            if (c >= currM && c < nextM && (h + 4) >= currH && (h + 4) < nextH && (w + 2) >=
currW && (w + 2) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 4][t3 + 2] * K4d(m, c, 4, 2);
            else
                y[bmhw] += x4d(b, c, (h + 4), (w + 2)) * K4d(m, c, 4, 2);

            if (c >= currM && c < nextM && (h + 4) >= currH && (h + 4) < nextH && (w + 3) >=
currW && (w + 3) < nextW)
```

```
                y[bmhw] += subTile[c - currM][t2 + 4][t3 + 3] * K4d(m, c, 4, 3);
            else
                y[bmhw] += x4d(b, c, (h + 4), (w + 3)) * K4d(m, c, 4, 3);

            if (c >= currM && c < nextM && (h + 4) >= currH && (h + 4) < nextH && (w + 4) >=
currW && (w + 4) < nextW)
                y[bmhw] += subTile[c - currM][t2 + 4][t3 + 4] * K4d(m, c, 4, 4);
            else
                y[bmhw] += x4d(b, c, (h + 4), (w + 4)) * K4d(m, c, 4, 4);
        }
      }
      __syncthreads();
   }

   #undef y4d
   #undef x4d
   #undef k4d

}
```

Final Milestone

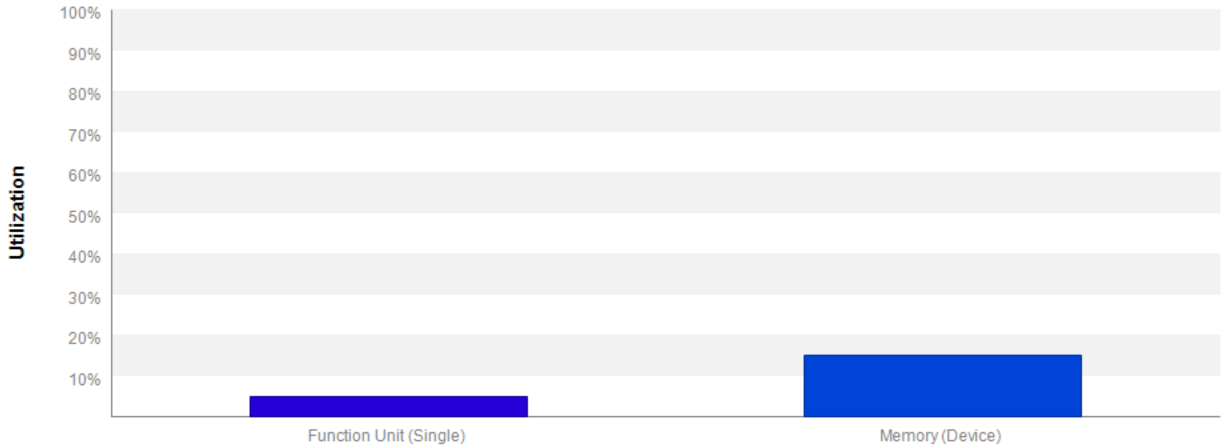
In the final milestone, we changed our implementation of kernels in the milestone 4 and applied 3 other optimizations.

Optimization 4: Unroll and shared-memory matrix multiply

We learn from the lecture slide that it is efficient to transform the convolution problem into matrix-multiplication problem by unrolling the input features and filters. We believe that this would be really helpful, since we can take advantage of the high efficiency of matrix-multiplication algorithm. Also, the use of shared-memory can also help improve memory throughput. After doing this optimization, the efficiency greatly increases as the NVVP analysis shown below.

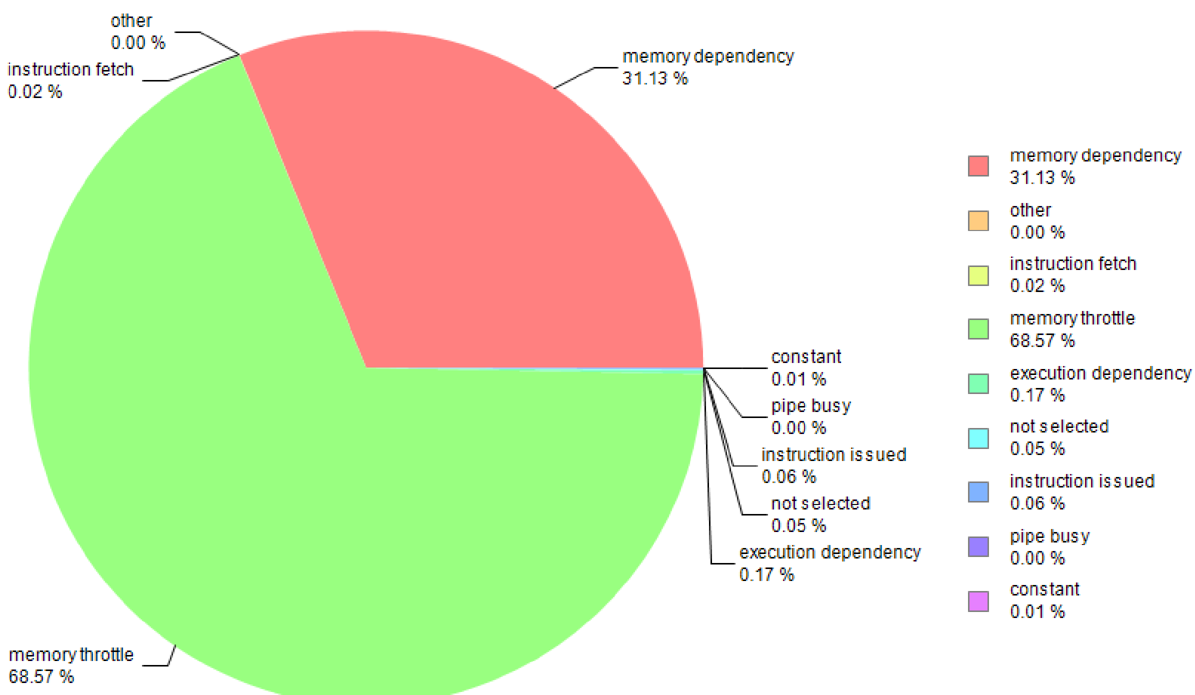
(Zhengqi, Licheng and Ruian worked on this optimization.)

NVVP Analysis
Note that this is generated from the code after optimization 5.
Before:

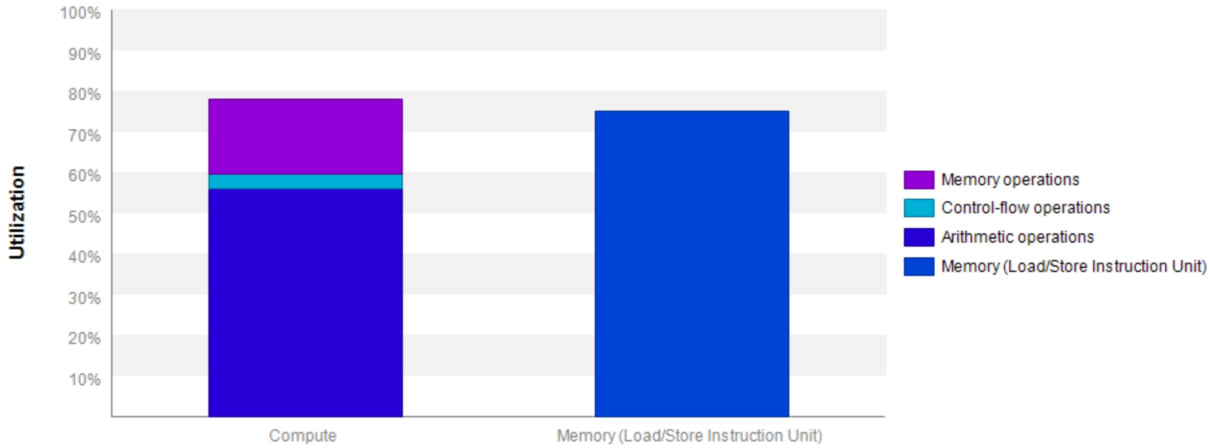


Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max

Sample distribution

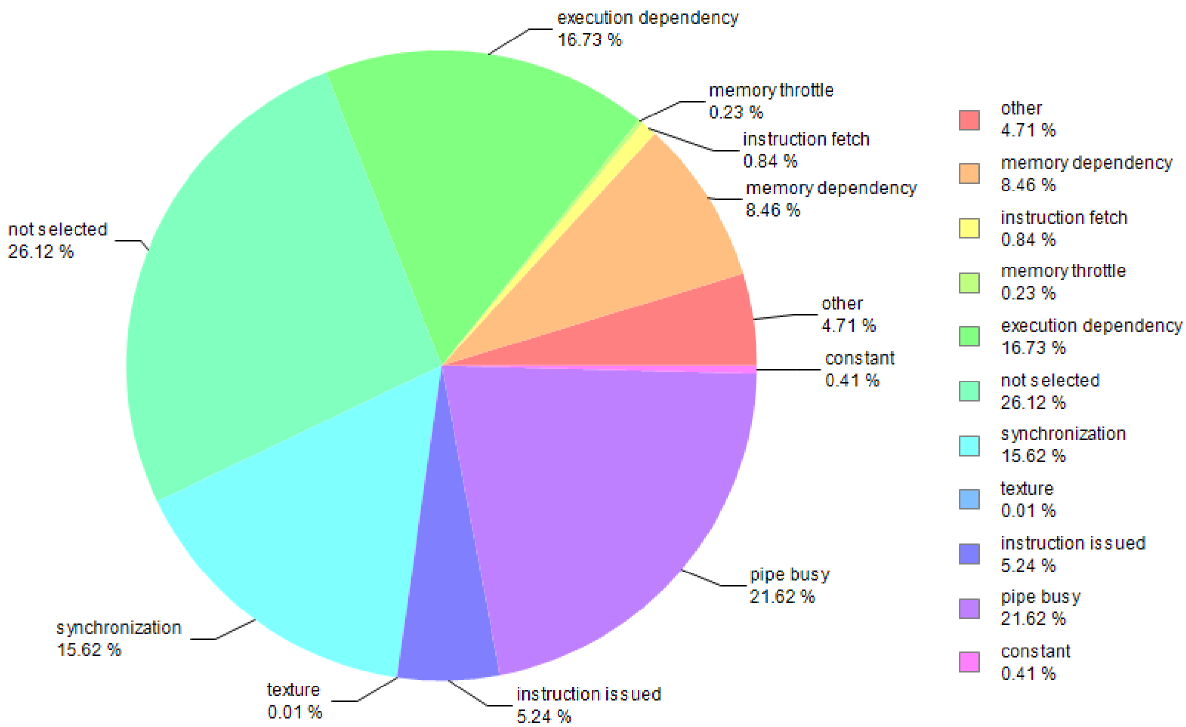


After:



Shared Loads	10973814	7,667.628 GB/s	
Shared Stores	1755359	1,226.505 GB/s	
Shared Total	12729173	8,894.134 GB/s	Idle Low Medium High Max

Sample distribution



Both compute and memory utilization greatly increases, and they are very close (both at 70%~80%) after the optimization. Moreover, the shared total also increases from zero to relatively high, which reflects the utilization of shared memory. Therefore, we can see from the sample distributions that before the optimization the bottleneck is memory throttle, but it is no longer an issue after the optimization. Now, each category has roughly the same stall proportions, and some of the stalls cannot be avoided (e.g., synchronization and execution dependency), which we think is ideal.

We also noticed that this optimization is more efficient than the shared-memory convolution in milestone 4. We suggest that this is because matrix-multiplication can use shared memory more efficiently than the regular convolution approach, and thus brings higher memory throughput.

CODE (kernel calls are omitted):

```
__global__ void unroll_kernel(const float *__restrict__ x, float
*x_unroll, const int B, const int M, const int C, const int H, const int
W, const int K)
{
    const int filterSize = K * K;
    int weightLength = C * filterSize;

    int numIter = ceil(weightLength / (1.0 * TILE_WIDTH));

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

#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1)
* (W) + i0]
#define x_unroll_3d(i2, i1, i0) x_unroll[(i2) * (weightLength * outCol)
+ (i1) * (outCol) + i0]

    int c, s, h_out, w_out, h_unroll, w_base, p, q;
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    int b = blockIdx.y * blockDim.y + threadIdx.y;

    if (t < C * outCol && b < B)
    {
        c = t / outCol;
        s = t % outCol;
        h_out = s / W_out;
        w_out = s % W_out;
        int w_unroll = s;
        w_base = c * filterSize;
        for (p = 0; p < K; p++)
        {
            for (q = 0; q < K; q++)
            {
                h_unroll = w_base + p * K + q;
                x_unroll_3d(b, h_unroll, w_unroll) = x4d(b, c, h_out +
p, w_out + q);
            }
        }
    }
#undef x4d
#undef x_unroll_3d
}

__global__ void matrixMultiplyShared(const float *__restrict__ A, float
```

```
*B, float *C, int batch,
                                      int numARows, int numAColumns,
                                      int numBRows, int numBColumns,
                                      int numCRows, int numCColumns)
{
    //@@ Insert code to implement matrix multiplication here
    //@@ You have to use shared memory for this MP
    __shared__ float subTileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileB[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
    float pValue = 0;

    for (int m = 0; m < (numAColumns - 1) / TILE_WIDTH + 1; m++)
    {
        if (row < numARows && (m * TILE_WIDTH + tx) < numAColumns)
        {
            subTileA[ty][tx] = A[row * numAColumns + m * TILE_WIDTH +
tx];
        }
        else
        {
            subTileA[ty][tx] = 0;
        }
        if (m * TILE_WIDTH + ty < numBRows && col < numBColumns && bz <
batch)
        {
            subTileB[ty][tx] = B[(bz * numBRows + m * TILE_WIDTH + ty) *
numBColumns + col];
        }
        else
        {
            subTileB[ty][tx] = 0;
        }
        __syncthreads();
        if (row < numCRows && col < numCColumns && bz < batch)
        {
            for (int k = 0; k < TILE_WIDTH; k++)
            {
                pValue += subTileA[ty][k] * subTileB[k][tx];
            }
        }
        __syncthreads();
    }
    if (row < numCRows && col < numCColumns && bz < batch)
        C[(bz * numCRows + row) * numCColumns + col] = pValue;
}
```

Optimization 5: Kernel fusion for unrolling and matrix multiplication

After the first optimization, we found that we were using two kernels while we could actually combine them into one. We believe kernel fusion is more efficient, because it reduces the overhead of starting up an extra kernel and saves the time of copying memory to and from an intermediate array for unrolled matrix. The improvement is tiny compared to that brought by unrolling and shared memory matrix multiply, but it still performs better.

According to nvprof and NVVP analysis, the amount of time performing compute reduced by about 0.2 ms and the amount of time required for memcpy reduced by about 0.1 ms.

(Zhengqi and Licheng worked on this optimization.)

We can see that both computation and memory copy time are reduced after the fusion of two kernels.

(We referenced the code from the Exam2 of FA2018.)

CODE:

```
__global__ void forward_kernel(float *__restrict__ y, const float
*__restrict__ x, const float *__restrict__ k, const int B, const int M,
const int C, const int H, const int W, const int K)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int col = bx * TILE_WIDTH + tx;
    int row = by * TILE_WIDTH + ty;

    const int filterSize = 25; // K * K = 25 (constant)
    int weightLength = C * filterSize;

    float acc = 0;

    int numIter = ceil(weightLength / (1.0 * TILE_WIDTH));

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

```
    const int outCol = H_out * W_out;

#define y4d(i3, i2, i1, i0) y[(i3) * (M * outCol) + (i2) * (outCol) +
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1)
* (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * filterSize) + (i2) *
(filterSize) + (i1) * (K) + i0]

    __shared__ float tileMatWUnroll[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileMatXUnroll[TILE_WIDTH][TILE_WIDTH];

    for (int i = 0; i < numIter; i++)
    {
        int tempCol = i * TILE_WIDTH + tx;
        int tempRow = i * TILE_WIDTH + ty;

        int W_m = row;
        int W_c = tempCol / filterSize;
        int W_h = (tempCol % filterSize) / K;
        int W_w = (tempCol % filterSize) % K;
        int X_b = bz;
        int X_c = tempRow / filterSize;
        int X_p = (tempRow % filterSize) / K;
        int X_q = (tempRow % filterSize) % K;
        int X_h = col / W_out;
        int X_w = col % W_out;


        if (tempCol < weightLength && row < M)
            tileMatWUnroll[ty][tx] = k4d(W_m, W_c, W_h, W_w);
        else
            tileMatWUnroll[ty][tx] = 0;
        if (tempRow < weightLength && col < outCol)
            tileMatXUnroll[ty][tx] = x4d(X_b, X_c, (X_h + X_p), (X_w +
X_q));
        else
            tileMatXUnroll[ty][tx] = 0;

        __syncthreads();

        acc += tileMatWUnroll[ty][0] * tileMatXUnroll[0][tx];
        acc += tileMatWUnroll[ty][1] * tileMatXUnroll[1][tx];
        acc += tileMatWUnroll[ty][2] * tileMatXUnroll[2][tx];
        acc += tileMatWUnroll[ty][3] * tileMatXUnroll[3][tx];
        acc += tileMatWUnroll[ty][4] * tileMatXUnroll[4][tx];
        acc += tileMatWUnroll[ty][5] * tileMatXUnroll[5][tx];
        acc += tileMatWUnroll[ty][6] * tileMatXUnroll[6][tx];
        acc += tileMatWUnroll[ty][7] * tileMatXUnroll[7][tx];
        acc += tileMatWUnroll[ty][8] * tileMatXUnroll[8][tx];
        acc += tileMatWUnroll[ty][9] * tileMatXUnroll[9][tx];
        acc += tileMatWUnroll[ty][10] * tileMatXUnroll[10][tx];
        acc += tileMatWUnroll[ty][11] * tileMatXUnroll[11][tx];
```

```
        acc += tileMatWUnroll[ty][12] * tileMatXUnroll[12][tx];
        acc += tileMatWUnroll[ty][13] * tileMatXUnroll[13][tx];
        acc += tileMatWUnroll[ty][14] * tileMatXUnroll[14][tx];
        acc += tileMatWUnroll[ty][15] * tileMatXUnroll[15][tx];
        acc += tileMatWUnroll[ty][16] * tileMatXUnroll[16][tx];
        acc += tileMatWUnroll[ty][17] * tileMatXUnroll[17][tx];
        acc += tileMatWUnroll[ty][18] * tileMatXUnroll[18][tx];
        acc += tileMatWUnroll[ty][19] * tileMatXUnroll[19][tx];
        acc += tileMatWUnroll[ty][20] * tileMatXUnroll[20][tx];
        acc += tileMatWUnroll[ty][21] * tileMatXUnroll[21][tx];
        acc += tileMatWUnroll[ty][22] * tileMatXUnroll[22][tx];
        acc += tileMatWUnroll[ty][23] * tileMatXUnroll[23][tx];
        acc += tileMatWUnroll[ty][24] * tileMatXUnroll[24][tx];
        acc += tileMatWUnroll[ty][25] * tileMatXUnroll[25][tx];
        acc += tileMatWUnroll[ty][26] * tileMatXUnroll[26][tx];
        acc += tileMatWUnroll[ty][27] * tileMatXUnroll[27][tx];
        acc += tileMatWUnroll[ty][28] * tileMatXUnroll[28][tx];
        acc += tileMatWUnroll[ty][29] * tileMatXUnroll[29][tx];
        acc += tileMatWUnroll[ty][30] * tileMatXUnroll[30][tx];
        acc += tileMatWUnroll[ty][31] * tileMatXUnroll[31][tx];

        __syncthreads();
    }
    int Y_b = bz;
    int Y_m = row;
    int Y_h = col / W_out;
    int Y_w = col % W_out;
    if (row < M && col < outCol)
    {
        y4d(Y_b, Y_m, Y_h, Y_w) = acc;
    }

#undef y4d
#undef x4d
#undef k4d
}
```

Optimization 6: Sweeping various parameters to find best values and miscellaneous

We have removed some redundant calculations within our kernels (Including the recalculation of indices and extra access for clearing out shared memory) and then focus on finding the best variables. It’s conceivable that this is going to have an impact on the runtime because it determines the size of the shared memory directly. As we increase the size of shared memory, presumably the time we spend for global memory access would decrease resulting in a better running time.

TILE_WIDTH = 8

Op Time: 0.022167

Op Time: 0.074366

Shared Memory

Shared Loads	11084619	6,234.757 GB/s	
Shared Stores	1767445	994.133 GB/s	
Shared Total	12852064	7,228.891 GB/s	Idle Low Medium High Max

L2 Cache

Reads	1094522	153.909 GB/s	
Writes	3921616	551.447 GB/s	
Total	5016138	705.356 GB/s	Idle Low Medium High Max

Unified Cache

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	6403435	900.434 GB/s	
Global Stores	3921600	551.445 GB/s	
Texture Reads	12894779	7,252.917 GB/s	
Unified Total	23219814	8,704.795 GB/s	Idle Low Medium High Max

TILE_WIDTH = 16

Op Time: 0.019408

Op Time: 0.056708

Shared Memory

Shared Loads	10968206	7,791.259 GB/s	
Shared Stores	877044	623.008 GB/s	
Shared Total	11845250	8,414.267 GB/s	Idle Low Medium High Max

L2 Cache

Reads	455704	80.927 GB/s	
Writes	1634416	290.252 GB/s	
Total	2090120	371.179 GB/s	Idle Low Medium High Max

Unified Cache

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2810578	499.123 GB/s	
Global Stores	1634400	290.249 GB/s	
Texture Reads	11689854	8,303.881 GB/s	
Unified Total	16134832	9,093.253 GB/s	Idle Low Medium High Max

TILE_WIDTH = 32

Op Time: 0.039544

Op Time: 0.059331

Shared Memory

Shared Loads	21119028	7,411.425 GB/s	
Shared Stores	886336	311.047 GB/s	
Shared Total	22005364	7,722.472 GB/s	Idle Low Medium High Max

L2 Cache

Reads	372323	32.665 GB/s	
Writes	735616	64.539 GB/s	
Total	1107939	97.204 GB/s	Idle Low Medium High Max

Unified Cache

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2382746	209.048 GB/s	
Global Stores	735600	64.537 GB/s	
Texture Reads	22471539	7,886.069 GB/s	
Unified Total	25589885	8,159.654 GB/s	Idle Low Medium High Max

TILE_WIDTH = 64
This is impossible given our block dimension is TILE_WIDTH*TILE_WIDTH and there are only 1024 threads at max per block. I have also tried thread coarsening to improve shared memory usage but then the bottleneck becomes the size of shared memory and a larger power of 2 would exceed the shared memory limit of the device by experiments.

Eventually, as TILE_WIDTH = 16 produces the best performance given the op time we observed and the shared memory bandwidth usage. Because of the best total runtime and the higher shared memory usage, that’s the dimension for our final code.

(Ruian and Licheng worked on this optimization.)

	No optimization	Best optimization (unroll + matrix-multiplication + parameter tuning/miscellaneous)
optime	5.771741 24.169380 Total: 29.941	0.017531 0.059203 Total: 0.0767

Optimization 7: An advanced matrix multiplication algorithm (register-tiled)

To Further improve our runtime, we tried to implement the register-tiled matrix multiplication in CUDA.

TILE_WIDTH_M = 32, TILE_WIDTH_N = 16:
Op Time: 0.030843
Op Time: 0.040050
Correctness: 0.7653 Model: ece408

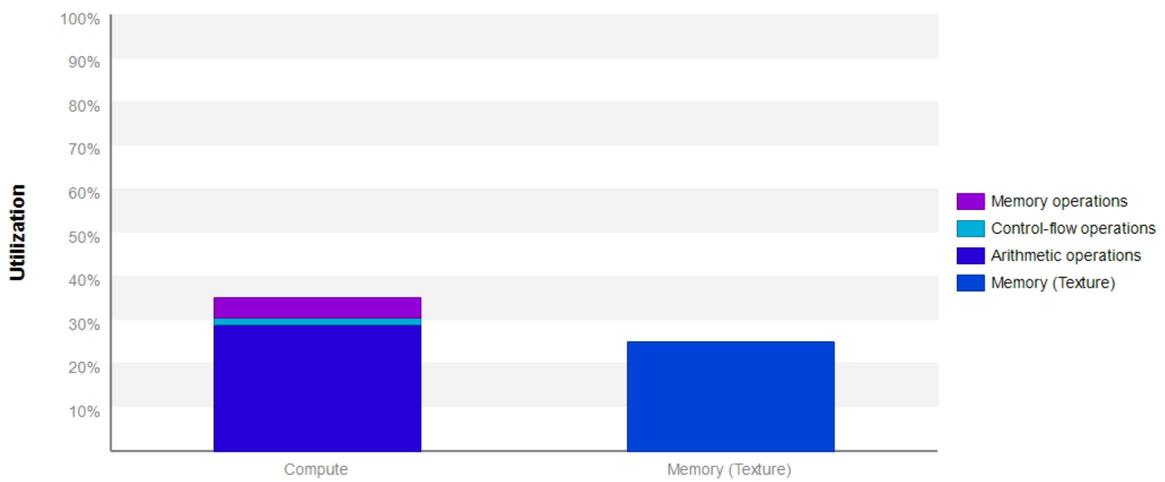
To clarify, the first op time has dimensions:
B: 10000
M: 12
C: 1

H: 70
W: 70
K: 5

The second op time has dimensions:
B: 10000
M: 24
C: 12
H: 33
W: 33
K: 5

Notice that our griDim and blockDim are set to follows:
gridDim(ceil(H_out * W_out / (1.0 * TILE_WIDTH_N)), ceil(M / (1.0 * TILE_WIDTH_M)), B);
blockDim(TILE_WIDTH_M, 1, 1);

Comparing with shared memory matrix multiplication version, we see that the time for the first set of convolution dimensions is slower while the time for the second set of convolution dimensions is faster. This is understandable given the first M is rather small and setting **TILE_WIDTH_M = 32** is clearly an overkill and creates a relatively unnecessary overhead.



Shared Memory

Shared Loads	5777344	2,318.342 GB/s	
Shared Stores	356954	143.239 GB/s	
Shared Total	6134298	2,461.581 GB/s	Idle Low Medium High Max

We can also see that the utilization of computation and the shared memory usage has reduced. This is understandable as we now used less shared memory and more registers.

We have also experimented with different **TILE_WIDTH_M and TILE_WIDTH_N** as well as different levels of **thread coarsening**:

TILE_WIDTH_M = 16, TILE_WIDTH_N = 8:
Op Time: 0.031925
Op Time: 0.095862
Correctness: 0.7653 Model: ece408

TILE_WIDTH_M = 32, TILE_WIDTH_N = 8:
Op Time: 0.031073
Op Time: 0.059261
Correctness: 0.7653 Model: ece408

TILE_WIDTH_M = 64, TILE_WIDTH_N = 32:
Op Time: 0.033002
Op Time: 0.048903
Correctness: 0.7653 Model: ece408

Nothing is as good as **TILE_WIDTH_M = 32, TILE_WIDTH_N = 16**. The reason being we need M to be large in order to enable parallelism in general, and we want N to be large in order to use the shared memory . But on the one hand the dimension limits us and too much thread coarsening would still decrease the benefits of parallelism. **TILE_WIDTH_M = 32, TILE_WIDTH_N = 16** achieves the best tradeoff between all these factors by testing. It's possible that maybe we can find another dimension mapping to achieve higher parallelism and thus achieve better performance but we didn't go as far as that.

(Licheng worked on this optimization.)

CODE:

```
const int TILE_WIDTH_M = 32;
const int TILE_WIDTH_N = 16;
const int STEPS = TILE_WIDTH_M / TILE_WIDTH_N;
__global__ void forward_kernel(float *__restrict__ y, const float
*__restrict__ x, const float *__restrict__ k, const int B, const int M,
const int C, const int H, const int W, const int K)
{

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int row = by * TILE_WIDTH_M + tx;
    const int filterSize = 25; // K * K = 25 (constant)
    int weightLength = C * filterSize;
    int numIter = ceil(weightLength / (1.0 * STEPS));
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    const int outCol = H_out * W_out;

#define y4d(i3, i2, i1, i0) y[(i3) * (M * outCol) + (i2) * (outCol) +
```

```
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1)
* (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * filterSize) + (i2) *
(filterSize) + (i1) * (K) + i0]

    float tileMatWUnroll[STEPS] = {};
    float acc[TILE_WIDTH_N] = {};
    __shared__ float tileMatXUnroll[STEPS][TILE_WIDTH_N];

    for (int i = 0; i < numIter; i++)
    {
        for (int j = 0; j < STEPS; j++)
        {
            int tempCol = i * STEPS + j;
            int W_m = row;
            int W_c = tempCol / filterSize;
            int W_h = (tempCol % filterSize) / K;
            int W_w = (tempCol % filterSize) % K;
            if (tempCol < weightLength && row < M)
                tileMatWUnroll[j] = k4d(W_m, W_c, W_h, W_w);
            else
                tileMatWUnroll[j] = 0;
        }

        int load_ty = tx / TILE_WIDTH_N;
        int load_tx = tx % TILE_WIDTH_N;
        int col = bx * TILE_WIDTH_N + load_tx;
        int tempRow = i * STEPS + load_ty;
        int X_b = bz;
        int X_c = tempRow / filterSize;
        int X_p = (tempRow % filterSize) / K;
        int X_q = (tempRow % filterSize) % K;
        int X_h = col / W_out;
        int X_w = col % W_out;
        if (tempRow < weightLength && col < outCol)
            tileMatXUnroll[load_ty][load_tx] = x4d(X_b, X_c, (X_h +
X_p), (X_w + X_q));
        else
            tileMatXUnroll[load_ty][load_tx] = 0;

        __syncthreads();

        for (int j = 0; j < TILE_WIDTH_N; j++)
        {
            /*Number of steps */
            acc[j] += tileMatWUnroll[0] * tileMatXUnroll[0][j];
            acc[j] += tileMatWUnroll[1] * tileMatXUnroll[1][j];
        }
        __syncthreads();
    }

    for (int j = 0; j < TILE_WIDTH_N; j++)
```

```
    {
        int col = bx * TILE_WIDTH_N + j;
        int Y_b = bz;
        int Y_m = row;
        int Y_h = col / W_out;
        int Y_w = col % W_out;
        if (row < M && col < outCol)
        {
            y4d(Y_b, Y_m, Y_h, Y_w) = acc[j];
        }
    }

#undef y4d
#undef x4d
#undef k4d
}

template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const
mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float>
&w)
{
    // Extract the tensor dimensions into B,M,C,H,W,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 = w.shape_[3];

    // Set the kernel dimensions
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    dim3 gridDim(ceil(H_out * W_out / (1.0 * TILE_WIDTH_N)),
                 ceil(M / (1.0 * TILE_WIDTH_M)),
                 B);
    dim3 blockDim(TILE_WIDTH_M, 1, 1);

    // Call the kernel
    forward_kernel<<<gridDim, blockDim>>>(y.dptr_, x.dptr_, w.dptr_, B,
M, C, H, W, K);

    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
}
```

Optimization 8: Multiple kernel implementations for different layer sizes

It’s rather natural that we can combine the advantages of both implementations just by looking at the runtime where shared memory matrix multiplication has the best first Op Time and the register-tiled has the best second Op Time. We can thus take the best performance for different sizes and combine them into one implementation. Given the dimensions we have above, we decided to use M as the threshold value where if M < 13, the program would fall back to shared memory matrix multiplication and otherwise it would use register-tiling.
The resulting Op Time works to our expectations:

TILE_WIDTH_M = 32, TILE_WIDTH_N = 16, TILE_WIDTH=16:
Op Time: 0.019410
Op Time: 0.041837
Correctness: 0.7653 Model: ece408

And this is our version for final ranking.

(Zhengqi, Ruian and Licheng worked on this optimization.)

FINAL CODE:

```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

namespace mxnet
{
namespace op
{

const int TILE_WIDTH = 16;
__global__ void forward_kernel_1(float *__restrict__ y, const float
*__restrict__ x, const float *__restrict__ k, const int B, const int M,
const int C, const int H, const int W, const int K)
{

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int col = bx * TILE_WIDTH + tx;
    int row = by * TILE_WIDTH + ty;

    const int filterSize = 25; // K * K = 25 (constant)
    int weightLength = C * filterSize;

    float acc = 0;
```

```
    int numIter = ceil(weightLength / (1.0 * TILE_WIDTH));

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

#define y4d(i3, i2, i1, i0) y[(i3) * (M * outCol) + (i2) * (outCol) +
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1)
* (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * filterSize) + (i2) *
(filterSize) + (i1) * (K) + i0]

    __shared__ float tileMatWUnroll[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileMatXUnroll[TILE_WIDTH][TILE_WIDTH];

    for (int i = 0; i < numIter; i++)
    {
        int tempCol = i * TILE_WIDTH + tx;
        int tempRow = i * TILE_WIDTH + ty;

        int W_m = row;
        int W_c = tempCol / filterSize;
        int W_h = (tempCol % filterSize) / K;
        int W_w = (tempCol % filterSize) % K;
        int X_b = bz;
        int X_c = tempRow / filterSize;
        int X_p = (tempRow % filterSize) / K;
        int X_q = (tempRow % filterSize) % K;
        int X_h = col / W_out;
        int X_w = col % W_out;

        if (tempCol < weightLength && row < M)
            tileMatWUnroll[ty][tx] = k4d(W_m, W_c, W_h, W_w);
        else
            tileMatWUnroll[ty][tx] = 0;
        if (tempRow < weightLength && col < outCol)
            tileMatXUnroll[ty][tx] = x4d(X_b, X_c, (X_h + X_p), (X_w +
X_q));
        else
            tileMatXUnroll[ty][tx] = 0;

        __syncthreads();

        acc += tileMatWUnroll[ty][0] * tileMatXUnroll[0][tx];
        acc += tileMatWUnroll[ty][1] * tileMatXUnroll[1][tx];
        acc += tileMatWUnroll[ty][2] * tileMatXUnroll[2][tx];
        acc += tileMatWUnroll[ty][3] * tileMatXUnroll[3][tx];
        acc += tileMatWUnroll[ty][4] * tileMatXUnroll[4][tx];
        acc += tileMatWUnroll[ty][5] * tileMatXUnroll[5][tx];
        acc += tileMatWUnroll[ty][6] * tileMatXUnroll[6][tx];
        acc += tileMatWUnroll[ty][7] * tileMatXUnroll[7][tx];
```

```
        acc += tileMatWUnroll[ty][8] * tileMatXUnroll[8][tx];
        acc += tileMatWUnroll[ty][9] * tileMatXUnroll[9][tx];
        acc += tileMatWUnroll[ty][10] * tileMatXUnroll[10][tx];
        acc += tileMatWUnroll[ty][11] * tileMatXUnroll[11][tx];
        acc += tileMatWUnroll[ty][12] * tileMatXUnroll[12][tx];
        acc += tileMatWUnroll[ty][13] * tileMatXUnroll[13][tx];
        acc += tileMatWUnroll[ty][14] * tileMatXUnroll[14][tx];
        acc += tileMatWUnroll[ty][15] * tileMatXUnroll[15][tx];

        __syncthreads();
    }
    int Y_b = bz;
    int Y_m = row;
    int Y_h = col / W_out;
    int Y_w = col % W_out;
    if (row < M && col < outCol)
    {
        y4d(Y_b, Y_m, Y_h, Y_w) = acc;
    }

#undef y4d
#undef x4d
#undef k4d
}

const int TILE_WIDTH_M = 32;
const int TILE_WIDTH_N = 16;
const int STEPS = TILE_WIDTH_M / TILE_WIDTH_N;
__global__ void forward_kernel_2(float *__restrict__ y, const float
*__restrict__ x, const float *__restrict__ k, const int B, const int M,
const int C, const int H, const int W, const int K)
{

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int row = by * TILE_WIDTH_M + tx;
    const int filterSize = 25; // K * K = 25 (constant)
    int weightLength = C * filterSize;
    int numIter = ceil(weightLength / (1.0 * STEPS));
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    const int outCol = H_out * W_out;

#define y4d(i3, i2, i1, i0) y[(i3) * (M * outCol) + (i2) * (outCol) +
(i1) * (W_out) + i0]
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1)
* (W) + i0]
#define k4d(i3, i2, i1, i0) k[(i3) * (C * filterSize) + (i2) *
(filterSize) + (i1) * (K) + i0]

    float tileMatWUnroll[STEPS] = {};
```

```
    float acc[TILE_WIDTH_N] = {};
    __shared__ float tileMatXUnroll[STEPS][TILE_WIDTH_N];

    for (int i = 0; i < numIter; i++)
    {
        for (int j = 0; j < STEPS; j++)
        {
            int tempCol = i * STEPS + j;
            int W_m = row;
            int W_c = tempCol / filterSize;
            int W_h = (tempCol % filterSize) / K;
            int W_w = (tempCol % filterSize) % K;
            if (tempCol < weightLength && row < M)
                tileMatWUnroll[j] = k4d(W_m, W_c, W_h, W_w);
            else
                tileMatWUnroll[j] = 0;
        }

        int load_ty = tx / TILE_WIDTH_N;
        int load_tx = tx % TILE_WIDTH_N;
        int col = bx * TILE_WIDTH_N + load_tx;
        int tempRow = i * STEPS + load_ty;
        int X_b = bz;
        int X_c = tempRow / filterSize;
        int X_p = (tempRow % filterSize) / K;
        int X_q = (tempRow % filterSize) % K;
        int X_h = col / W_out;
        int X_w = col % W_out;
        if (tempRow < weightLength && col < outCol)
            tileMatXUnroll[load_ty][load_tx] = x4d(X_b, X_c, (X_h +
X_p), (X_w + X_q));
        else
            tileMatXUnroll[load_ty][load_tx] = 0;

        __syncthreads();


        for (int j = 0; j < TILE_WIDTH_N; j++)
        {
            /*Number of steps */
            acc[j] += tileMatWUnroll[0] * tileMatXUnroll[0][j];
            acc[j] += tileMatWUnroll[1] * tileMatXUnroll[1][j];
        }
        __syncthreads();
    }

    for (int j = 0; j < TILE_WIDTH_N; j++)
    {
        int col = bx * TILE_WIDTH_N + j;
        int Y_b = bz;
        int Y_m = row;
        int Y_h = col / W_out;
        int Y_w = col % W_out;
```

```
        if (row < M && col < outCol)
        {
            y4d(Y_b, Y_m, Y_h, Y_w) = acc[j];
        }
    }

#undef y4d
#undef x4d
#undef k4d
}

template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const
mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4, float>
&w)
{
    // Extract the tensor dimensions into B,M,C,H,W,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 = w.shape_[3];

    // Set the kernel dimensions
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    if (M < 13)
    {
        dim3 gridDim_1(ceil(H_out * W_out / (1.0 * TILE_WIDTH)),
                       ceil(M / (1.0 * TILE_WIDTH)),
                       B);
        dim3 blockDim_1(TILE_WIDTH, TILE_WIDTH, 1);
        forward_kernel_1<<<gridDim_1, blockDim_1>>>(y.dptr_, x.dptr_,
w.dptr_, B, M, C, H, W, K);
    }
    else
    {
        dim3 gridDim_2(ceil(H_out * W_out / (1.0 * TILE_WIDTH_N)),
                       ceil(M / (1.0 * TILE_WIDTH_M)),
                       B);
        dim3 blockDim_2(TILE_WIDTH_M, 1, 1);

        // Call the kernel
        forward_kernel_2<<<gridDim_2, blockDim_2>>>(y.dptr_, x.dptr_,
w.dptr_, B, M, C, H, W, K);
    }

    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
}
```