ECE 408/CS 483 Final Project

Team: elegant_and_easygoing_boys
Team members:
Licheng Luo (ll6)
Zhengqi Fang (zf4)
Ruian Pan (ruianp2)
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)

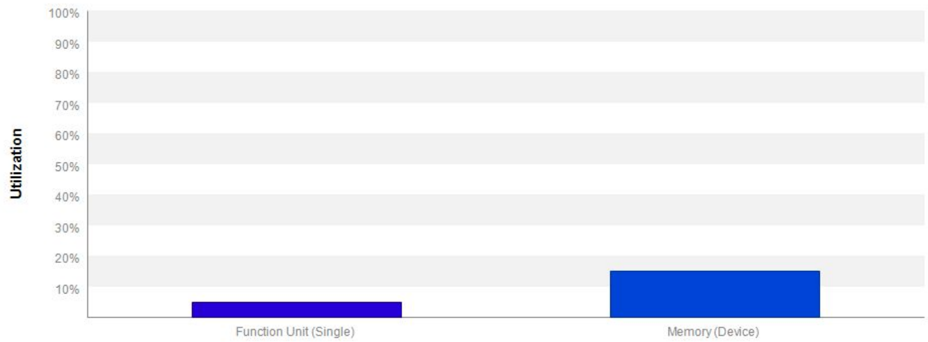
Note that 99.6% of the Compute is the forward_kernel.

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:

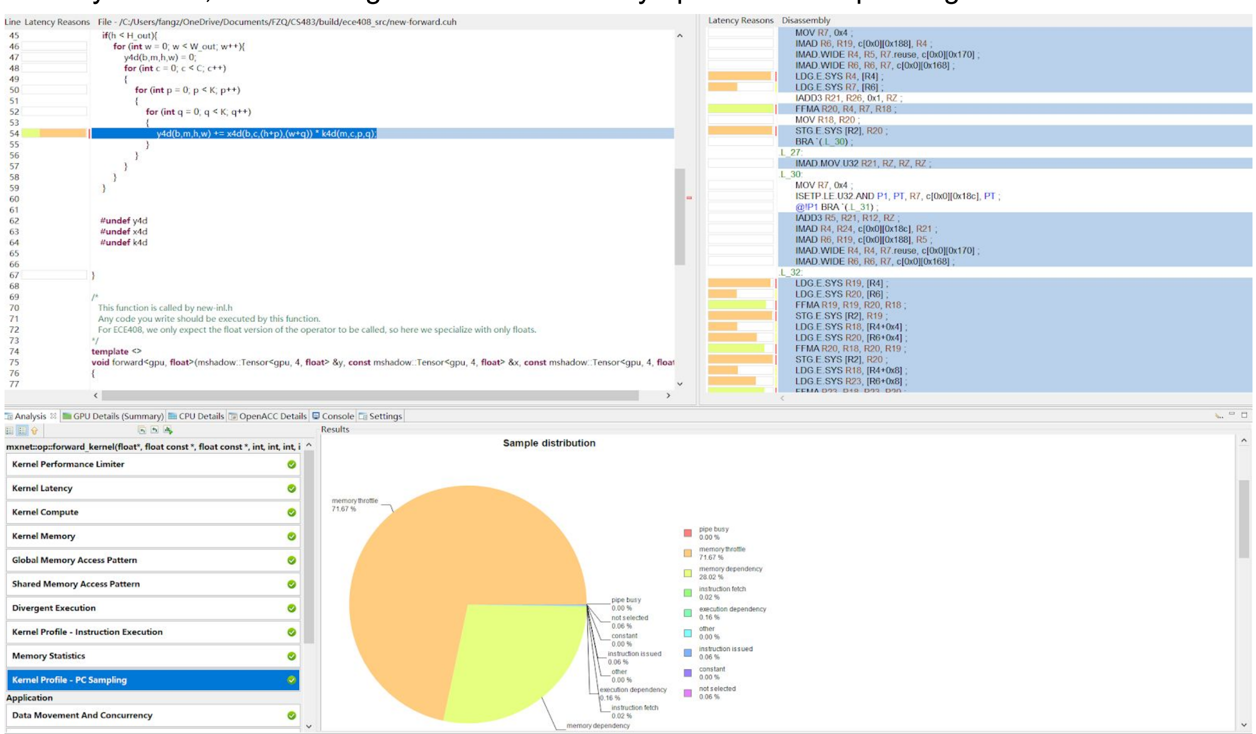
i Kernel Performance Is Bound By Instruction And Memory Latency

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

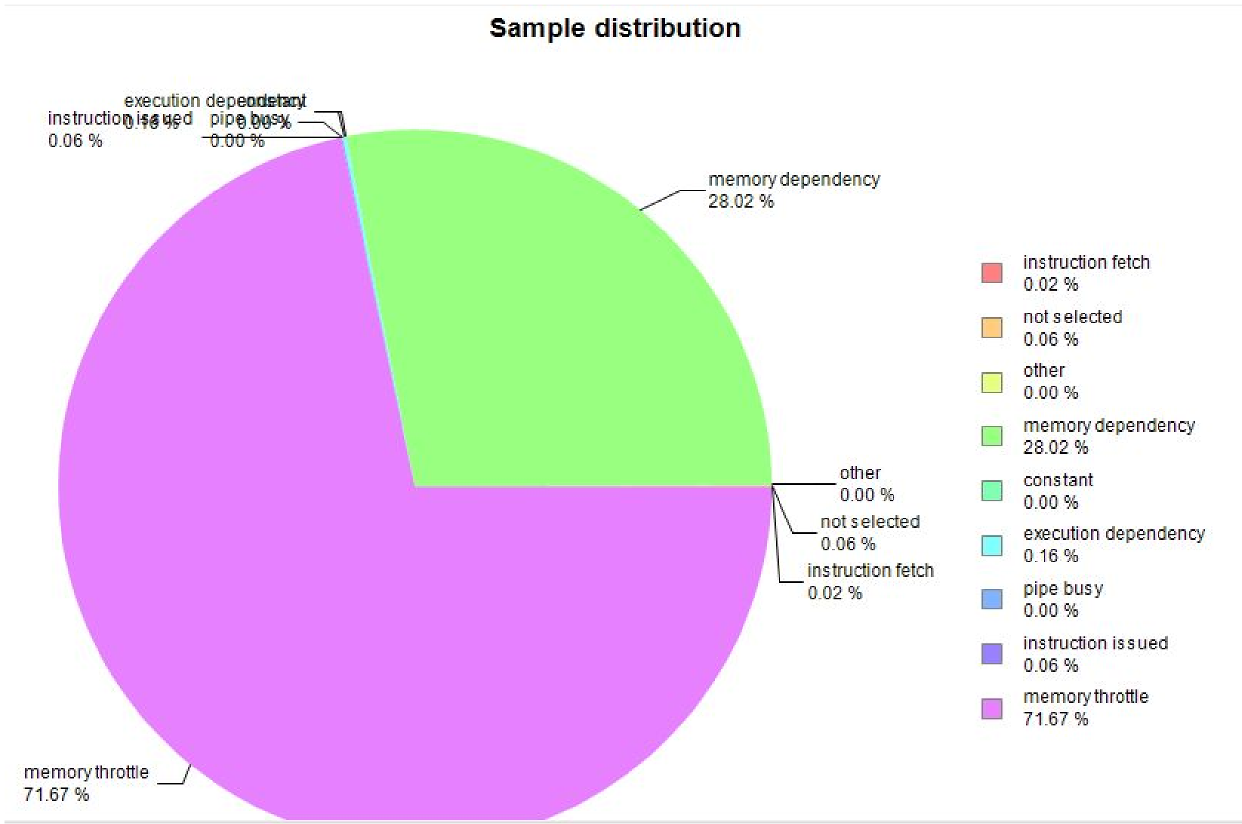
The three optimizations we use are **Shared Memory convolution, Weight matrix (kernel values) in constant memory** and **Tuning with restrict and loop unrolling.**

Shared Memory convolution:

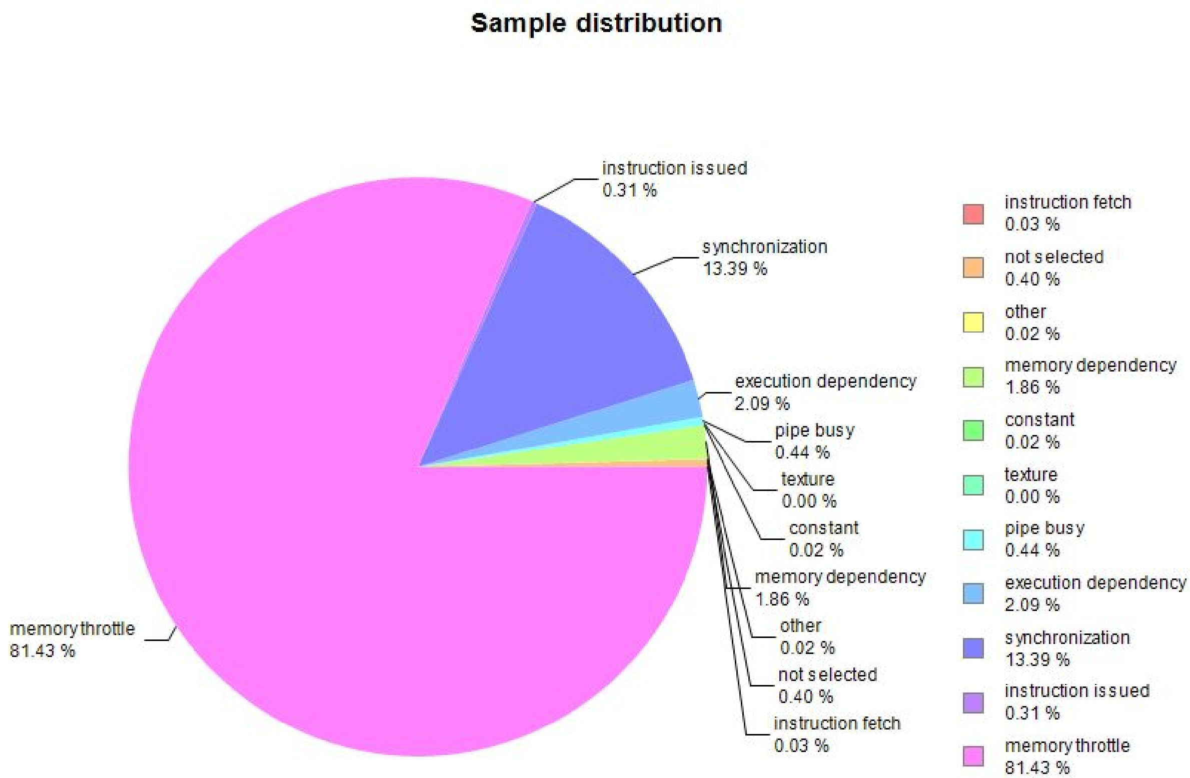
The shared memory convolution can greatly reduce the global memory access of x array. By reusing the tiling region, and keeping a reduced number of global access, it improves the performance of our code. (See the red portion of the code for our usage).

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.

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.

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>

Data Size 10000	Milestone 3 (No	Milestone 4 (With all

	optimization)	optimizations)
Profiling using usr/bin/time	User: 25.01, System: 9.50, Elapsed: 0:31.01	User: 9.40, System: 4.16, Elapsed: 0:10.08

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.

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

}
```