

EECS 498 Final Project

Team member: Zhongqian Duan (duanzq), Ruipu Li (liruipu)

Optimization1: Unroll + shared-memory Matrix multiply (Abandoned)

We first implemented the suggested optimization **Unroll + shared-memory Matrix multiply**.

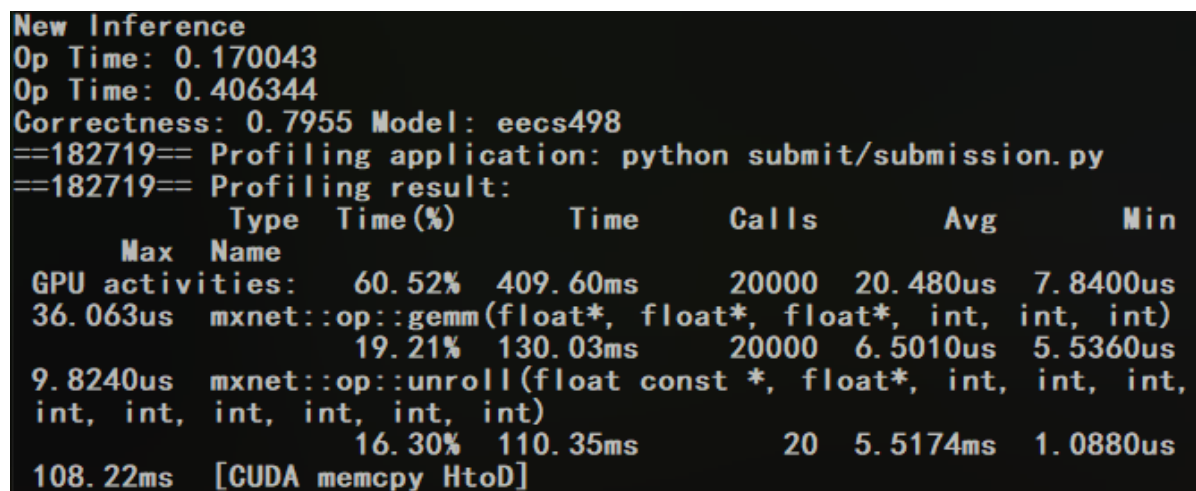
The unrolling step does the index mapping and stores the unrolling matrix. And this step can transfer the convolution problem into the matrix multiply problem. The kernel for **Unroll** imitates the code shown in lecture 19, page 34. Note that in the slide, the values for `w_unroll` and `h_unroll` are incorrect. The correct value should be

```
int w_unroll = h_out * w_out + w_out;
int h_unroll = c * K * K + p * K + q;
```

In the **shared-memory matrix multiply** step, we used the kernel in the HW3 Tiled Matrix Multiplication. Both the weight matrix and unrolled `x` are loaded into shared memory before computation.

Initially, we set the `TILE_WIDTH = 32` (for matrix multiply) and `BLOCK_SIZE = 128` (for unroll). The run time for both layers is 0.170043 and 0.406344 separately.

The screenshot of the output given `TILE_WIDTH = 32` and `BLOCK_SIZE = 128`:



```
New Inference
Op Time: 0.170043
Op Time: 0.406344
Correctness: 0.7955 Model: eeecs498
==182719== Profiling application: python submit/submission.py
==182719== Profiling result:
           Type  Time(%)   Time     Calls   Avg       Min
    Max  Name
GPU activities:  60.52%  409.60ms   20000  20.480us  7.8400us
36.063us  mxnet::op::gemm(float*, float*, float*, int, int, int)
           19.21%  130.03ms   20000   6.5010us  5.5360us
9.8240us  mxnet::op::unroll(float const *, float*, int, int, int,
int, int, int, int, int, int)
           16.30%  110.35ms     20    5.5174ms  1.0880us
108.22ms  [CUDA memcpy HtoD]
```

The next optimization we tried was **Sweeping various parameters to find best values**. We use grid search to find the best hyperparameter: `TILE_WIDTH` and `BLOCK_SIZE`. The result is shown in the following table. We found that when `TILE_WIDTH = 16` and `BLOCK_SIZE = 128`, the speed is fastest. But it still did not achieve the goal (under 0.3 seconds for both layers combined).

TILE_WIDTH	BLOCK_SIZE	OP Time 1	OP Time 2
16	64	0.148216	0.339553
16	128	0.136755	0.289244
16	256	0.140186	0.293229
16	512	0.146108	0.305463
32	64	0.160920	0.453823
32	128	0.170043	0.406344
32	256	0.171340	0.407562
32	512	0.170779	0.418037

Since there are too many outputs in the table, we decided to put only one screenshot in the report. The screenshot of the output given TILE_WIDTH = 16 and BLOCK_SIZE = 128:

```
New Inference
Op Time: 0.136755
Op Time: 0.289244
Correctness: 0.7955 Model: eeecs498
==157427== Profiling application: python submit/submission.py
==157427== Profiling result:
      Type  Time(%)    Time    Calls    Avg      Min
  Max Name
GPU activities:  50.01%  263.43ms   20000  13.171us  5.4080us
25.152us mxnet::op::gemm(float*, float*, float*, int, int, int)
          24.09%  126.89ms   20000   6.3440us  5.6320us
9.5040us mxnet::op::unroll(float const *, float*, int, int, int,
int, int, int, int, int)
          20.80%  109.56ms     20    5.4781ms  1.1830us
107.42ms [CUDA memcpy HtoD]
```

Optimization2: Kernel fusion for unrolling and matrix-multiplication (Abandoned)

Then we tried to implement **Kernel fusion for unrolling and matrix-multiplication**. The implementation detail is that instead of launching a kernel to unroll the input x and having a separate matrix to store the unrolled x, we directly calculate the correct index when loading the input into the shared memory in the matrix multiplication step.

Unexpectedly, we found that the run time became larger in the second convolution layer. The possible reason is that getting the correct index requires lots of calculations, so it is better to launch the unrolling kernel and do it concurrently.

The screenshot of the output:

```

New Inference
Op Time: 0.081066
Op Time: 0.368898
Correctness: 0.7955 Model: eecs498
==200445== Profiling application: python submit/submission.py
==200445== Profiling result:
           Type  Time(%)   Time     Calls       Avg       Min
    Max  Name
GPU activities:  75.74%  432.61ms    20000  21.630us  6.5920us
39.008us mxnet::op::matrixMultiplyShared_Unroll(float*, float*,
float*, int, int, int, int, int, int)
           19.51%  111.46ms        20   5.5728ms  1.1200us
109.29ms [CUDA memcpy HtoD]

```

Optimization3: Shared Memory convolution

Since global memory read and global memory write are quite slow, this optimization utilizes the shared memory to have several threads share the local version of data. Both the weight matrix and input data x are loaded into shared memory before computation.

The forward kernel for **Shared Memory convolution** imitates the code shown in lecture 19, page 28. We also added more boundary checks to avoid invalid memory access. We found two mistakes in the slide.

In the step of loading tiles from input data x into shared memory, the correct code should be

```
x_shared[i - h_base, j - w_base] = x4d(b, c, i, j);
```

instead of

```
x_shared[i - h_base, j - w_base] = x4d(b, c, h, w);
```

And in the step of doing convolution, , the correct code should be

```
acc += X_shared[h0 + p, w0 + q] * w_shared[p, q];
```

instead of

```
acc += X_shared[h + p, w + q] * w_shared[p, q];
```

The screenshot of the output:

```

New Inference
Op Time: 0.120604
Op Time: 0.237765
Correctness: 0.7955 Model: eecs498
==47777== Profiling application: python submit/submission.py
==47777== Profiling result:
           Type  Time(%)   Time     Calls       Avg       Min
    Max  Name
GPU activities:  72.24%  358.26ms         2  179.13ms  120.56ms
237.70ms mxnet::op::ConvLayerForward(float*, float const *, floa
t const *, int, int, int, int, int, int)
           22.31%  110.65ms        20   5.5324ms  1.0880us
108.51ms [CUDA memcpy HtoD]

```

Optimization4: Weight matrix (kernel values) in constant memory

Since the weight matrix keeps unchanged during the computation, it is feasible to load it into constant memory. And when the weight matrix is loaded into constant memory, there is no need to load it into shared memory, which can accelerate the kernel.

The size of weight matrix is MCK^2 . For the first convolution layer, $M = 12, C = 1, K = 7$. For the second convolution layer, $M = 24, C = 12, K = 7$.

We set the weight matrix in constant memory as

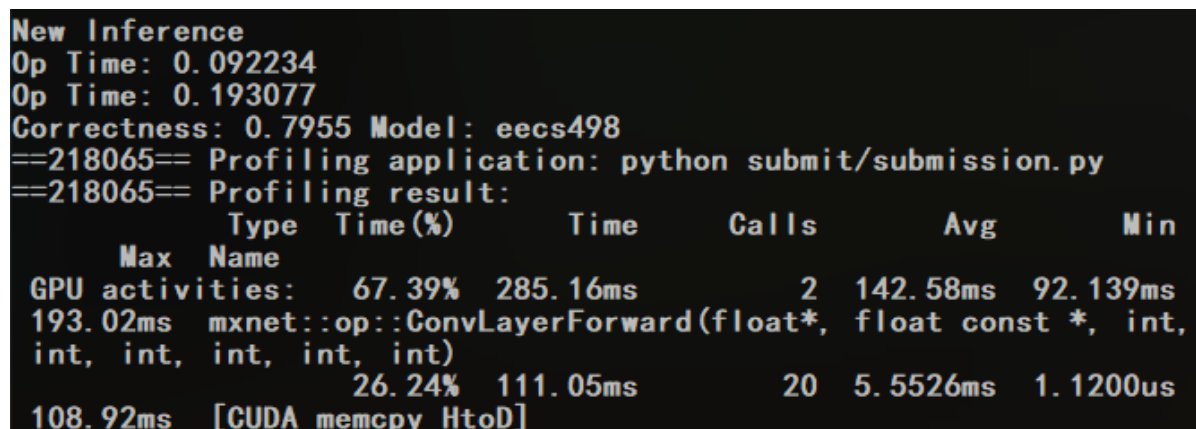
```
__constant__ float deviceKernel[14112];
```

,where $14112 = 24 \times 12 \times 7^2$.

The forward kernel is almost the same as the one in **Shared Memory convolution**. The main difference is that it saves the step to load the weight matrix into shared memory, and it directly uses the weight matrix in constant memory when calculating convolution.

After that optimization, it finally achieves the goal (under 0.3 seconds for both layers combined)!

The screenshot of the output:



```
New Inference
Op Time: 0.092234
Op Time: 0.193077
Correctness: 0.7955 Model: eecs498
==218065== Profiling application: python submit/submission.py
==218065== Profiling result:
           Type  Time(%)      Time       Calls         Avg         Min
    Max  Name
GPU activities:  67.39%  285.16ms           2    142.58ms   92.139ms
193.02ms mxnet::op::ConvLayerForward(float*, float const *, int,
int, int, int, int, int)
           26.24%  111.05ms          20     5.5526ms   1.1200us
108.92ms [CUDA memcpy HtoD]
```

Appendix

The source code for Unroll + shared-memory Matrix multiply

```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

#define TILE_WIDTH 16
#define BLOCK_SIZE 128

namespace mxnet
{
    namespace op
```

```

{
__global__ void gemm(float *A, float *B, float *C, int numRows, int
numAColumns, int numBColumns) {
    __shared__ float subTileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileB[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float temp = 0;

    for(int m = 0; m < ceil((float)numAColumns/TILE_WIDTH); m++){
        if(Row < numRows && m * TILE_WIDTH + tx < numAColumns)
            subTileA[ty][tx] = A[Row * numAColumns + m * TILE_WIDTH + tx]; //A[Row][m
* TILE_WIDTH + tx]
        else
            subTileA[ty][tx] = 0;

        if(m * TILE_WIDTH + ty < numAColumns && Col < numBColumns)
            subTileB[ty][tx] = B[(m * TILE_WIDTH + ty) * numBColumns + Col]; //B[m *
TILE_WIDTH + ty][Col]
        else
            subTileB[ty][tx] = 0;

        __syncthreads();
        for(int k = 0; k < TILE_WIDTH; k++){
            temp += subTileA[ty][k] * subTileB[k][tx];
        }
        __syncthreads();
    }
    if(Row < numRows && Col < numBColumns)
        C[Row * numBColumns + Col] = temp;
}

__global__ void unroll(const float *x, float *x_unroll, int b, const int C,
const int H, const int W, const int K, int H_out, int W_out, int H_unroll, int
w_unroll) {
#define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) +
i0]
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < C * w_unroll) {
        int c = idx / w_unroll;
        int s = idx % w_unroll;
        int h_out = s / W_out;
        int w_out = s % W_out;
        int w_unroll = h_out * W_out + w_out;
        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q++) {
                int h_unroll = c * K * K + p * K + q;
                x_unroll[h_unroll * w_unroll + w_unroll] = x4d(b, c, h_out + p,
w_out + q);
            }
        }
    }
}
#undef x4d

```

```

}

void ConvLayerForward(float *y, float *x, float *k, int B, int M, int C, int H,
int W, int K){
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    int H_unroll = C * K * K;
    int W_unroll = H_out * W_out;

    //int numRows = M;
    //int numAColumns = H_unroll;
    //int numBColumns = W_unroll;
    //int numBRows = numAColumns;
    //int numCRows = numRows;
    //int numCColumns = numBColumns;
    dim3 dimGrid1(ceil((float)H_unroll * W_unroll/BLOCK_SIZE), 1, 1);
    dim3 dimBlock1(BLOCK_SIZE, 1, 1);
    dim3 dimGrid2(ceil((float)W_unroll/TILE_WIDTH), ceil((float)M/TILE_WIDTH),
1);
    dim3 dimBlock2(TILE_WIDTH, TILE_WIDTH, 1);

    float *x_unroll;
    cudaMalloc((void **) &x_unroll, H_unroll * W_unroll * sizeof(float));

    for(int b = 0; b < B; b++){
        unroll<<<dimGrid1, dimBlock1>>>(x, x_unroll, b, C, H, W, K, H_out,
W_out, H_unroll, W_unroll);
        gemm<<<dimGrid2, dimBlock2>>>(k, x_unroll, &y[b*M*H_out*W_out], M,
H_unroll, W_unroll);
    }
    cudaFree(x_unroll);
}

/*
    This function is called by new-inl.h
    Any code you write should be executed by this function.
    We only expect the float version of the operator to be called, so here we
specialize with only floats.
*/
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)
{

    // // Use mxnet's CHECK_EQ to do assertions.
    // CHECK_EQ(0, 1)

    const int B = x.shape_[0];
    const int M = y.shape_[1]; // num_filter
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = w.shape_[3];

    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
    ConvLayerForward(y.dptr_, x.dptr_, w.dptr_, B, M, C, H, W, K);
}

```

```

    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
}

/*
    This tells mxnet how to do an op when it's not a float.
    This is not used in the project
*/
template <typename gpu, typename DType>
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
{
    assert(0 && "No forward implementation for other datatypes needed");
}
}
}

#endif

```

The source code for Kernel fusion for unrolling and matrix-multiplication

```

#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

#define TILE_WIDTH 16
#define BLOCK_SIZE 128

namespace mxnet
{
    namespace op
    {
        __global__ void matrixMultiplyShared_Unroll(float *kernel, float *x, float *y,
                                                    int numRows, int numAColumns, int
numBColumns,
                                                    int C, int K, int H, int W) {
            __shared__ float subTileA[TILE_WIDTH][TILE_WIDTH];
            __shared__ float subTileB[TILE_WIDTH][TILE_WIDTH];
            int bx = blockIdx.x; int by = blockIdx.y;
            int tx = threadIdx.x; int ty = threadIdx.y;

            int Row = by * TILE_WIDTH + ty;
            int Col = bx * TILE_WIDTH + tx;

            int w_out = W - K + 1;
            int w = Col % w_out;
            int h = Col / w_out;
            float temp = 0;

            for(int m = 0; m < ceil((float)numAColumns/TILE_WIDTH); m++){
                if(Row < numRows && m * TILE_WIDTH + tx < numAColumns)
                    subTileA[ty][tx] = kernel[Row * numAColumns + m * TILE_WIDTH + tx];
                //A[Row][m * TILE_WIDTH + tx]
            }
            else
                subTileA[ty][tx] = 0;
        }
    }
}

```

```

int h_unroll = m * TILE_WIDTH + ty;
if(h_unroll < numAColumns && Col < numBColumns){
    int q = h_unroll % K;
    h_unroll /= K;
    int p = h_unroll % K;
    int c = h_unroll / K;
    subTileB[ty][tx] = x[c * (H * W) + (h+p) * (W) + w+q];
}
else
    subTileB[ty][tx] = 0;

__syncthreads();
for(int k = 0; k < TILE_WIDTH; k++){
    temp += subTileA[ty][k] * subTileB[k][tx];
}
__syncthreads();
}
if(Row < numRows && Col < numBColumns)
    y[Row * numBColumns + Col] = temp;
}

void convLayer_forward(float *y, float *x, float *k, int B, int M, int C, int H,
int W, int K){
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    int H_unroll = C * K * K;
    int W_unroll = H_out * W_out;

    dim3 dimGrid(ceil((float)W_unroll/TILE_WIDTH), ceil((float)M/TILE_WIDTH),
1);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    for(int b = 0; b < B; b++){
        matrixMultiplyShared_Unroll<<<dimGrid, dimBlock>>>(k, &x[b*C*H*W],
&y[b*M*H_out*W_out], M, H_unroll, W_unroll,C,K,H,W);
    }
}

/*
    This function is called by new-inl.h
    Any code you write should be executed by this function.
    We only expect the float version of the operator to be called, so here we
specialize with only floats.
*/
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)
{

    // // Use mxnet's CHECK_EQ to do assertions.
    // CHECK_EQ(0, 1)

    const int B = x.shape_[0];

```



```

const int M = y.shape_[1]; // num_filter
const int C = x.shape_[1];
const int H = x.shape_[2];
const int W = x.shape_[3];
const int K = w.shape_[3];

MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
convLayer_forward(y.dptr_, x.dptr_, w.dptr_, B, M, C, H, W, K);
MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
}

/*
This tells mxnet how to do an op when it's not a float.
This is not used in the project
*/
template <typename gpu, typename DType>
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
{
    assert(0 && "No forward implementation for other datatypes needed");
}
}
}
}

#endif

```

The source code for shared Memory convolution

```

#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

#define TILE_WIDTH 32

namespace mxnet
{
    namespace op
    {
        __global__ void ConvLayerForward(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)
        {

#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) k[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) +
i0]

            const int H_out = H - K + 1;
            const int W_out = W - K + 1;
            int w_grid = ceil((float)W_out/TILE_WIDTH);

```

```

int H_grid = ceil((float)H_out/TILE_WIDTH);

int X_TILE_WIDTH = TILE_WIDTH + K - 1;
extern __shared__ float shmem[];
float* sharedX = &shmem[0];
float* sharedW = &shmem[X_TILE_WIDTH * X_TILE_WIDTH];
int b = blockIdx.x;
int m = blockIdx.y;
int h0 = threadIdx.x;
int w0 = threadIdx.y;

int h_base = (blockIdx.z / W_grid) * TILE_WIDTH;
int w_base = (blockIdx.z % W_grid) * TILE_WIDTH;
int h = h_base + h0;
int w = w_base + w0;

float acc = 0.0;

for (int c = 0; c < C; c++) {
    if(h0 < K && w0 < K)
        sharedW[h0 * K + w0] = k4d(m, c, h0, w0);
    __syncthreads();

    for (int i = h; i < h_base + X_TILE_WIDTH; i += TILE_WIDTH) {
        for (int j = w; j < w_base + X_TILE_WIDTH; j += TILE_WIDTH) {
            if(i < H && j < W)
                sharedX[(i - h_base) * X_TILE_WIDTH + (j - w_base)] = x4d(b,
c, i, j);
            else
                sharedX[(i - h_base) * X_TILE_WIDTH + (j - w_base)] = 0;
        }
    }
    __syncthreads();

    for (int p = 0; p < K; p++) {
        for (int q = 0; q < K; q++) {
            if(h0 + p < X_TILE_WIDTH && w0 + q < X_TILE_WIDTH)
                acc += sharedX[(h0 + p) * X_TILE_WIDTH + (w0 + q)] *
sharedW[p * K + q];
        }
    }
    __syncthreads();
}
if(b < B && m < M && h < H_out && w < W_out)
    y4d(b, m, h, w) = acc;

#undef y4d
#undef x4d
#undef k4d
}
/*
    This function is called by new-inl.h
    Any code you write should be executed by this function.
    We only expect the float version of the operator to be called, so here we
    specialize with only floats.
*/
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)
{

    // // Use mxnet's CHECK_EQ to do assertions.
    // CHECK_EQ(0, 1)

    const int B = x.shape_[0];
    const int M = y.shape_[1]; // num_filter
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = w.shape_[3];

    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    int w_grid = ceil((float)W_out/TILE_WIDTH);
    int H_grid = ceil((float)H_out/TILE_WIDTH);

    int X_TILE_WIDTH = TILE_WIDTH + K - 1;
    size_t shared_size = sizeof(float) * (X_TILE_WIDTH*X_TILE_WIDTH+K*K);

    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 dimGrid(B, M, H_grid * W_grid);

    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
    ConvLayerForward<<<dimGrid, dimBlock, shared_size>>>(y.dptr_, x.dptr_,
w.dptr_, B, M, C, H, W, K);
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}

/*
    This tells mxnet how to do an op when it's not a float.
    This is not used in the project
*/
template <typename gpu, typename DType>
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
{
    assert(0 && "No forward implementation for other datatypes needed");
}
}
}

#endif

```

The source code for shared Memory convolution & Weight matrix (kernel values) in constant memory

```

#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#include <mxnet/base.h>

```

```

#define TILE_WIDTH 32

namespace mxnet
{
    namespace op
    {
        __constant__ float deviceKernel[14112];

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

#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) k[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) +
i0]
#define const_k4d(i3, i2, i1, i0) deviceKernel[(i3) * (C * K * K) + (i2) * (K *
K) + (i1) * (K) + i0]

            const int H_out = H - K + 1;
            const int W_out = W - K + 1;
            int w_grid = ceil((float)W_out/TILE_WIDTH);
            int h_grid = ceil((float)H_out/TILE_WIDTH);

            int X_TILE_WIDTH = TILE_WIDTH + K - 1;
            extern __shared__ float shmem[];
            float* sharedX = &shmem[0];
            //float* sharedW = &shmem[X_TILE_WIDTH * X_TILE_WIDTH];
            int b = blockIdx.x;
            int m = blockIdx.y;
            int h0 = threadIdx.x;
            int w0 = threadIdx.y;

            int h_base = (blockIdx.z / w_grid) * TILE_WIDTH;
            int w_base = (blockIdx.z % w_grid) * TILE_WIDTH;
            int h = h_base + h0;
            int w = w_base + w0;

            float acc = 0.0;

            for (int c = 0; c < C; c++) {
                for (int i = h; i < h_base + X_TILE_WIDTH; i += TILE_WIDTH) {
                    for (int j = w; j < w_base + X_TILE_WIDTH; j += TILE_WIDTH) {
                        if(i < H && j < W)
                            sharedX[(i - h_base) * X_TILE_WIDTH + (j - w_base)] = x4d(b,
c, i, j);
                        else
                            sharedX[(i - h_base) * X_TILE_WIDTH + (j - w_base)] = 0;
                    }
                }
                __syncthreads();

                for (int p = 0; p < K; p++) {
                    for (int q = 0; q < K; q++) {
                        if(h0 + p < X_TILE_WIDTH && w0 + q < X_TILE_WIDTH)

```

```

        //acc += sharedX[(h0 + p) * X_TILE_WIDTH + (w0 + q)] *
sharedw[p * K + q];
        acc += sharedX[(h0 + p) * X_TILE_WIDTH + (w0 + q)] *
const_k4d(m, c, p, q);
    }
}
__syncthreads();
}
if(b < B && m < M && h < H_out && w < W_out)
    y4d(b, m, h, w) = acc;

#undef y4d
#undef x4d
#undef k4d
#undef const_k4d
}
/*
    This function is called by new-inl.h
    Any code you write should be executed by this function.
    We only expect the float version of the operator to be called, so here we
    specialize with only floats.
*/
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)
{

    // // Use mxnet's CHECK_EQ to do assertions.
    // CHECK_EQ(0, 1)

    const int B = x.shape_[0];
    const int M = y.shape_[1]; // num_filter
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = w.shape_[3];

    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    int w_grid = ceil((float)W_out/TILE_WIDTH);
    int h_grid = ceil((float)H_out/TILE_WIDTH);

    int X_TILE_WIDTH = TILE_WIDTH + K - 1;
    size_t shared_size = sizeof(float) * (X_TILE_WIDTH*X_TILE_WIDTH);

    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 dimGrid(B, M, h_grid * w_grid);

    cudaMemcpyToSymbol(deviceKernel, (M*C*K*K) * sizeof(float));

    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
    ConvLayerForward<<<dimGrid, dimBlock, shared_size>>>(y.dptr_, x.dptr_, B, M,
C, H, W, K);
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}

/*

```

This tells mxnet how to do an op when it's not a float.

This is not used in the project

```
*/  
template <typename gpu, typename DType>  
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4,  
DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)  
{  
    assert(0 && "No forward implementation for other datatypes needed");  
}  
}  
}  
  
#endif
```