<u>Optimization 1:</u> Weight matrix (kernel values) in constant memory

- **Motivation & Benefits:** Since we know the weight matrix does not change through kernel execution and is small in size we can place it in constant memory. By placing the weights in constant memory this reduces global memory consumption which will improve performance.

```
__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_y, const float *host
{
    // Allocate memory and copy over the relevant data structures to the GPU
    cudaMalloc(device_x_ptr, B*C*H*W*sizeof(float));
    cudaMalloc(device_y_ptr, B*M*(W-K+1)*(H-K+1)*sizeof(float));
    cudaMalloc(device_k_ptr, M*C*K*K*sizeof(float));

    cudaMemcpy(*device_x_ptr, host_x, B*C*H*W*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(*device_k_ptr, host_k, M*C*K*K*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(filters, host_k, K*K*M*C*sizeof(float));
}
```

```
#define TILE_WIDTH 20
#define TILE_WIDTH2 17
#define FILTER_WIDTH 7

__constant__ float filters[7*7*16*4];

// each thread calculates 4 outputs
__global__ void conv_forward_kernel(float *y, const float* __restrict__ x, const float* k, const int B, const i
{

    // An example use of these macros:
    // float a = y4d(0,0,0,0)
    // y4d(0,0,0,0) = a
```

- **Effect on Runtime:** Achieved a runtime of about **33.32 ms**

<u>Optimization 2:</u> Tuning with restrict and loop unrolling

- **Motivation & Benefits:** Loop unrolling is a common compiler optimization to reduce overhead that comes with looping instructions. Additionally, since we know the values coming from the input feature maps in array **x** are going to be repeatedly used throughout the kernel execution we can aggressively cache the values using **__restrict__**.

```
__global__ void conv_forward_kernel2(float *y, const float* __restrict__ x, const f
const int W, const int K)
{
```

```
__global__ void conv_forward_kernel(float *y, const float* __restrict__ x, cons
const int W, const int K)
```

```
    #pragma unroll
    for(int i = 0; i < 8; i++) {
      #pragma unroll
      for(int j = 0; j < 8; j++) {
          int r = h+i, c = w+j;
          input[0][i][j] = shmem[0][r][c];
          input[1][i][j] = shmem[1][r][c];
          input[2][i][j] = shmem[2][r][c];
          input[3][i][j] = shmem[3][r][c];
      }
    }

    float accUpperLeft = 0.0f, accBottomRight = 0.0f, accBottomLeft = 0.0f, accTopRight = 0.0f;

   #pragma unroll
    for(int r = 0; r < FILTER_WIDTH; r++) {
      #pragma unroll
      for(int col = 0; col < FILTER_WIDTH; col++) {
          float zero = k4d(m, 0, r, col), one = k4d(m, 1, r, col), two = k4d(m, 2, r, col), three = k4d(m, 3, r, col);
```

```
    for(int r = 0; r < FILTER_WIDTH; r++) {
      #pragma unroll 7
      for(int col = 0; col < FILTER_WIDTH; col++) {
          acc += shmem[0][threadIdx.y+r][threadIdx.x+col] * k4d(m, 0, r, col);
          accNextB += shmem[1][threadIdx.y+r][threadIdx.x+col] * k4d(m, 0, r, col);
          third += shmem[2][threadIdx.y+r][threadIdx.x+col] * k4d(m, 0, r, col);
          fourth += shmem[3][threadIdx.y+r][threadIdx.x+col] * k4d(m, 0, r, col);
      }
    }
```

- **Effect on Runtime:** Achieved an improved runtime of about **18.83 ms**

<u>Optimization 3:</u> Multiple kernel implementations for different layer sizes

- **Motivation & Benefits:** The layers have different dimensions so it makes sense to have specialized layers to optimize for one case. This allows specific tuning (such as different tile sizes) to optimize each layer separately.

```
__host__ void GPUInterface::conv_forward_gpu(float *device_y, const float *device_x, const float *dev:
C, const int H, const int W, const int K)
{
    // Set the kernel dimensions and call the kernel
    if(M < 16) {
        dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);

        int Z = (int)ceil((W-K+1)*1.0/(TILE_WIDTH))*(int)ceil((H-K+1)*1.0/(TILE_WIDTH));

        dim3 gridDim(B/4,M,Z);

        conv_forward_kernel<<<gridDim, blockDim>>>(device_y, device_x, device_k, B, M, C, H, W, K);
    } else {
        dim3 blockDim(TILE_WIDTH2, TILE_WIDTH2, 1);

        int Z = (int)ceil((W-K+1)*1.0/(2*TILE_WIDTH2))*(int)ceil((H-K+1)*1.0/(2*TILE_WIDTH2));

        dim3 gridDim(B,M,Z);

        conv_forward_kernel2<<<gridDim, blockDim>>>(device_y, device_x, device_k, B, M, C, H, W, K);
    }
}
```

- **Effect on Runtime:** Achieved an improved runtime of about **13.24 ms**

Optimization 4: Using Shared Memory & Registers

- **Motivation & Benefits:** To reduce the global memory bandwidth consumption we can load into shared memory. In the second layer, I decided to do another optimization to rely on registers (each thread calculates 4 output values in this layer).

```
__shared__ float shmem[4][sharedWidth][sharedWidth];

int horizontalTiles = ceil((W_out)*1.0/TILE_WIDTH);

int b = 4*blockIdx.x, nextB=b+1, thirdB=b+2, fourthB=b+3, m = blockIdx.y;

int h_upperLeft = (blockIdx.z / (horizontalTiles))*TILE_WIDTH, w_upperLeft = (

int h = h_upperLeft + threadIdx.y, w = w_upperLeft + threadIdx.x;

int h_bottomRight = h_upperLeft + sharedWidth, w_bottomRight = w_upperLeft + sh

for(int i = h; i < h_bottomRight; i += TILE_WIDTH) {
   for(int j = w; j < w_bottomRight; j += TILE_WIDTH) {
       shmem[0][i - h_upperLeft][j - w_upperLeft] = x4d(b, 0, i, j);
       shmem[1][i - h_upperLeft][j - w_upperLeft] = x4d(nextB, 0, i, j);
       shmem[2][i - h_upperLeft][j - w_upperLeft] = x4d(thirdB, 0, i, j);
       shmem[3][i - h_upperLeft][j - w_upperLeft] = x4d(fourthB, 0, i, j);
   }
}
```

```
__shared__ float shmem[4][40][40];

int b = blockIdx.x, m = blockIdx.y, tx = 2*threadIdx.x, ty = 2*threadIdx

int h = ty, w = tx;

for(int i = threadIdx.y; i < 40; i += TILE_WIDTH2) {
   #pragma unroll
   for(int j = threadIdx.x; j < 40; j += TILE_WIDTH2) {
       shmem[0][i][j] = x4d(b, 0, i, j);
       shmem[1][i][j] = x4d(b, 1, i, j);
       shmem[2][i][j] = x4d(b, 2, i, j);
       shmem[3][i][j] = x4d(b, 3, i, j);
   }
}

__syncthreads();

// load into registers
float input[4][8][8];

#pragma unroll
for(int i = 0; i < 8; i++) {
   #pragma unroll
   for(int j = 0; j < 8; j++) {
       int r = h+i, c = w+j;
       input[0][i][j] = shmem[0][r][c];
       input[1][i][j] = shmem[1][r][c];
       input[2][i][j] = shmem[2][r][c];
       input[3][i][j] = shmem[3][r][c];
   }
}
```

- **Effect on Runtime:** Achieved an improved runtime of about **4.228 ms**