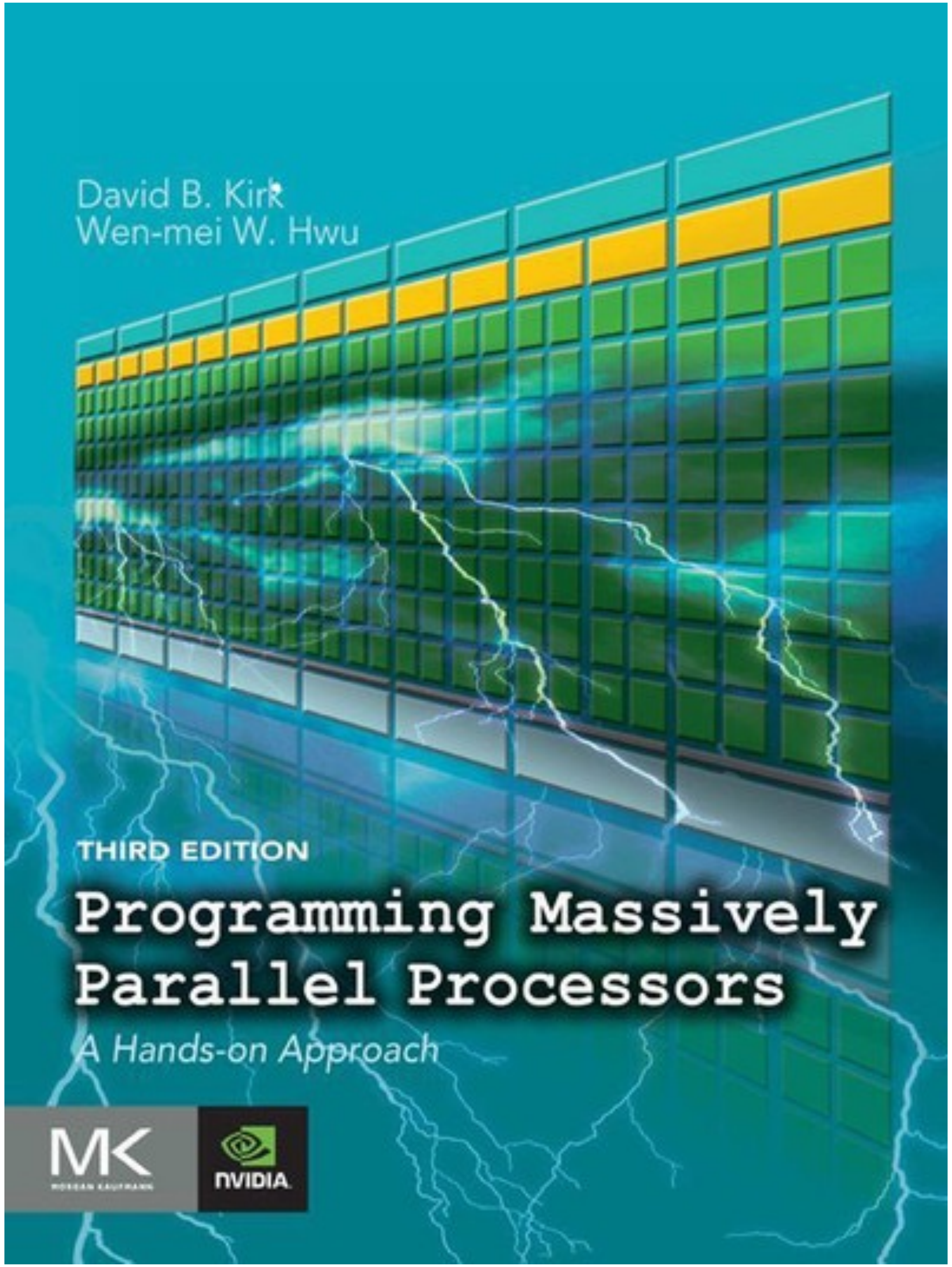
11868 LLM Systems
GPU Acceleration

Lei Li

Carnegie Mellon University
Language Technologies Institute



https://cmu.primo.exlibrisgroup.com/permalink/01CMU_INST/6lpsnm/alma991019904889504436

Today's Topic

- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

Today's Topic

- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

Simple Version of Matrix Multiplication

```
void matrix_multiply(float **a, float **b, float **c, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            c[i][j] = 0;
            for (int k = 0; k < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

complexity: $o(n^3)$

compute-to-global-memory-access: 1.0

```
__global__ void MatMulKernel(float *a, float *b, float *c, int N) {
    // Compute each thread's global row and col index -> output: (i, j)
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row >= N || col >= N) return;
    float Pvalue = 0.0;
    for (int k = 0; k < N; k++) {
        Pvalue += a[row * N + k] * b[k * N + col];
    }
    c[row * N + col] = Pvalue;
}
```

2 global memory access

vs

1 float add & 1 float mul

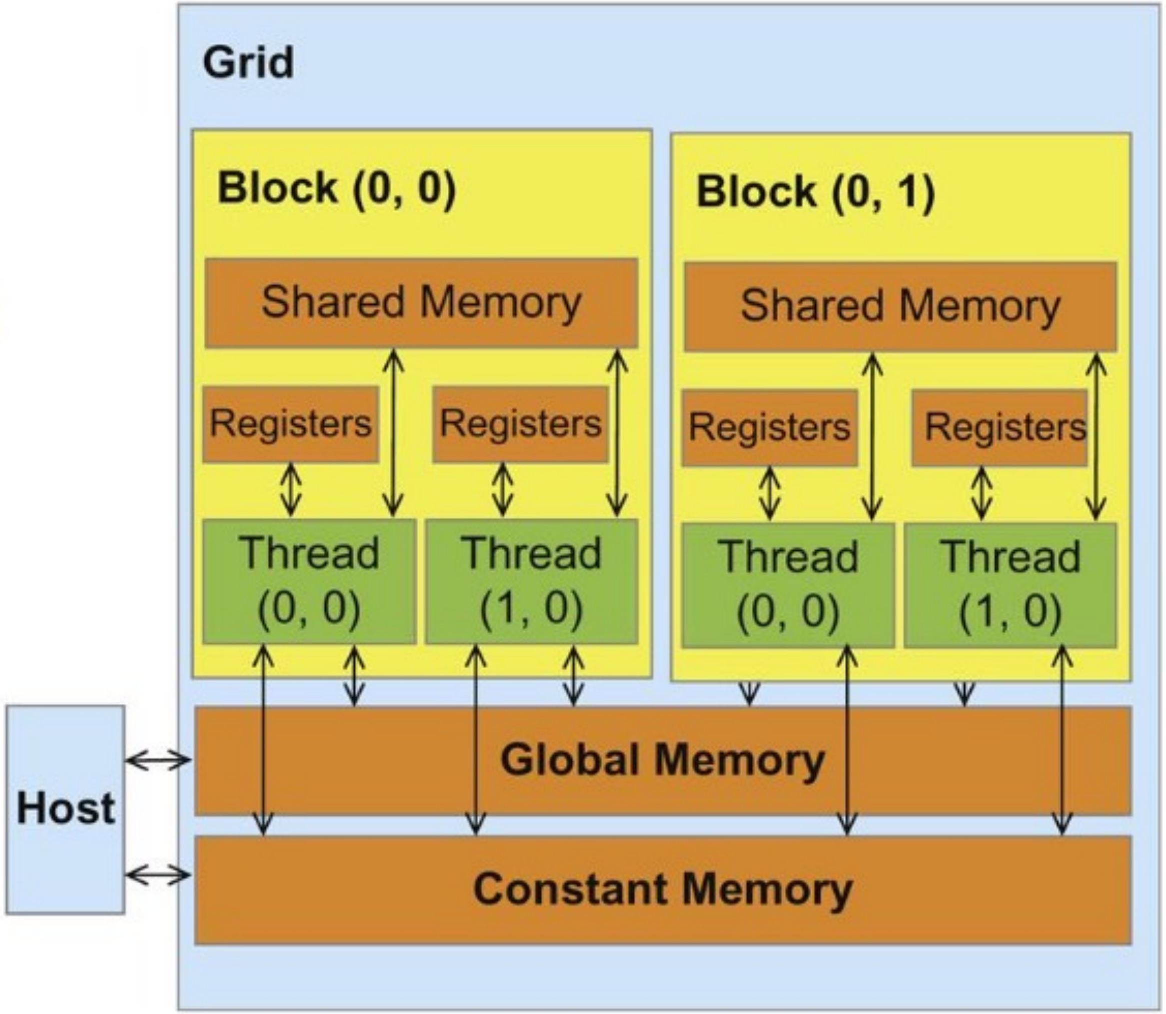
CUDA Device Memory Model

Device code can:

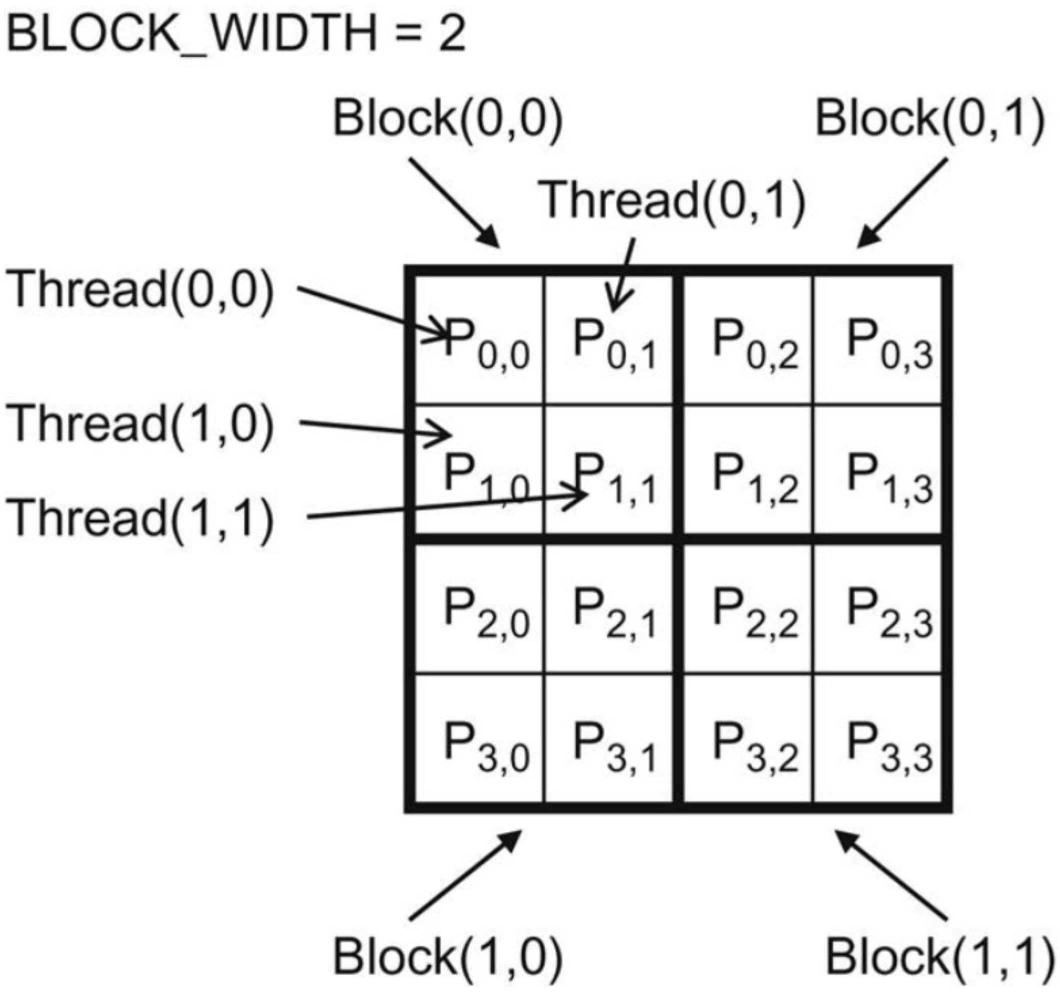
- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



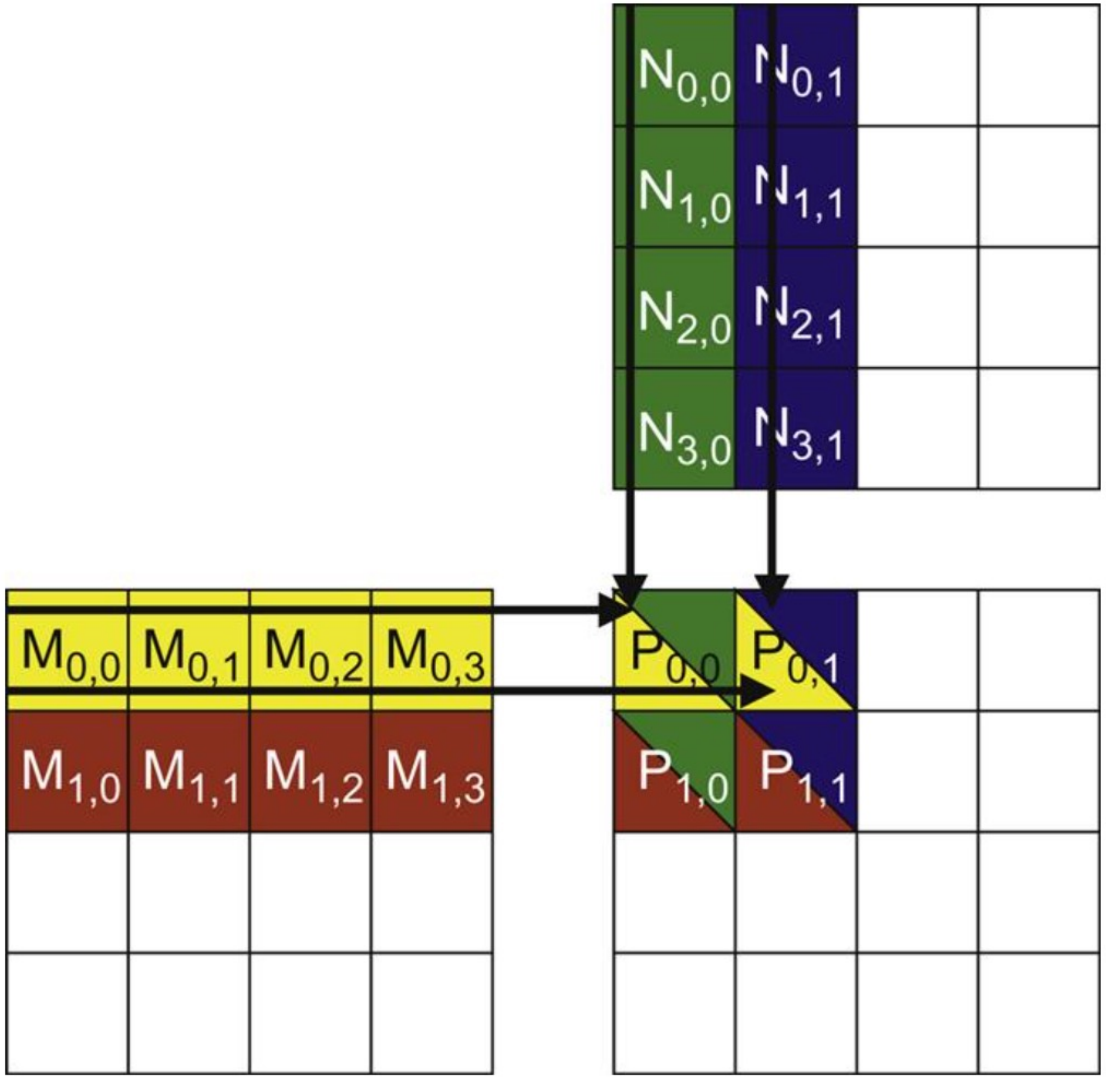
Tiling



- Each block has 4 threads.
- Every thread is responsible for calculating one element of P.
- There are four blocks in total.

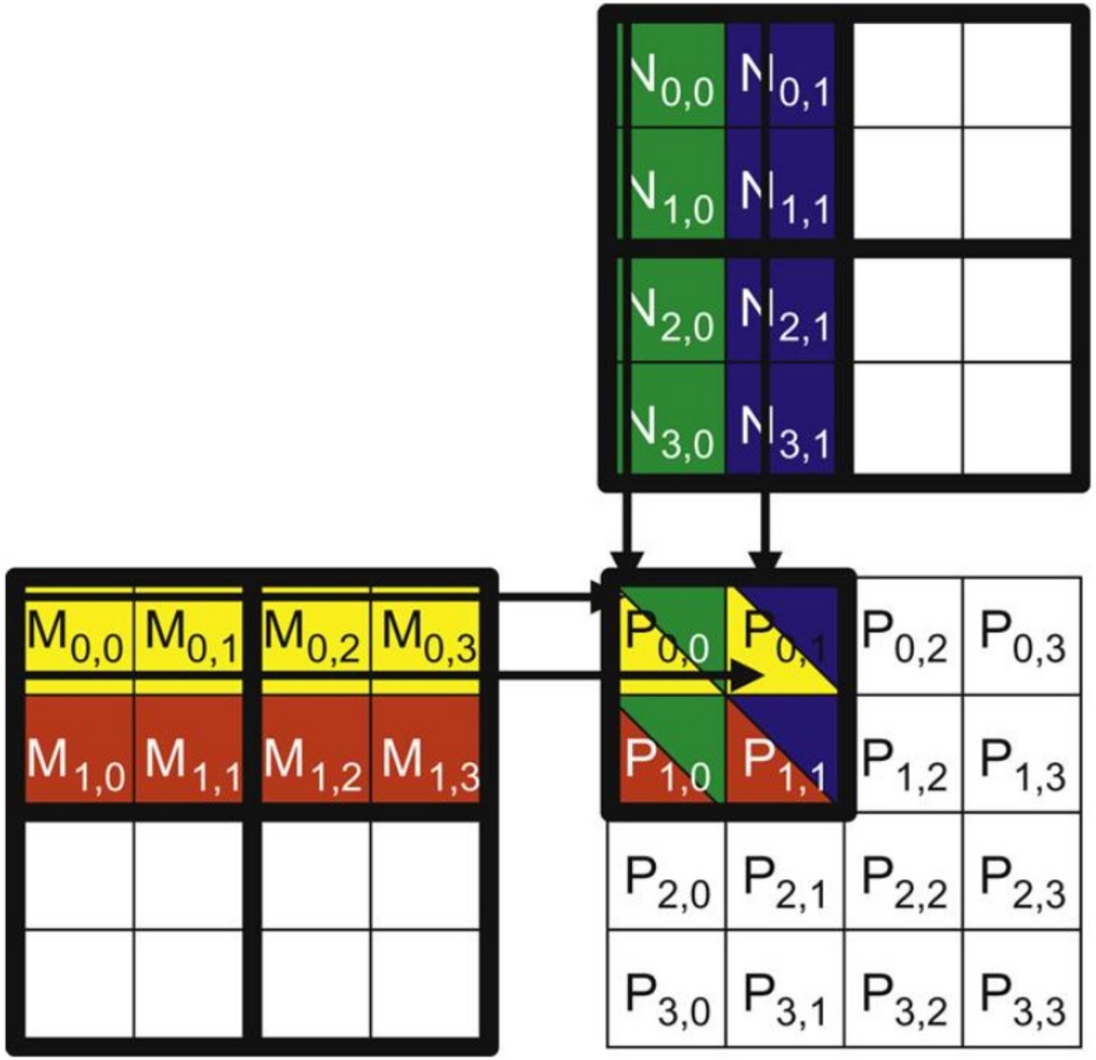
```
dim3 dimBlock(2, 2);
dim3 dimGrid(2, 2);
```

Tiling



- Each block has 4 threads.
- Every thread i has to load Mi,0-3 from global memory once.
- If thread0 and thread1 work together, then the # of accesses is reduced by ½.

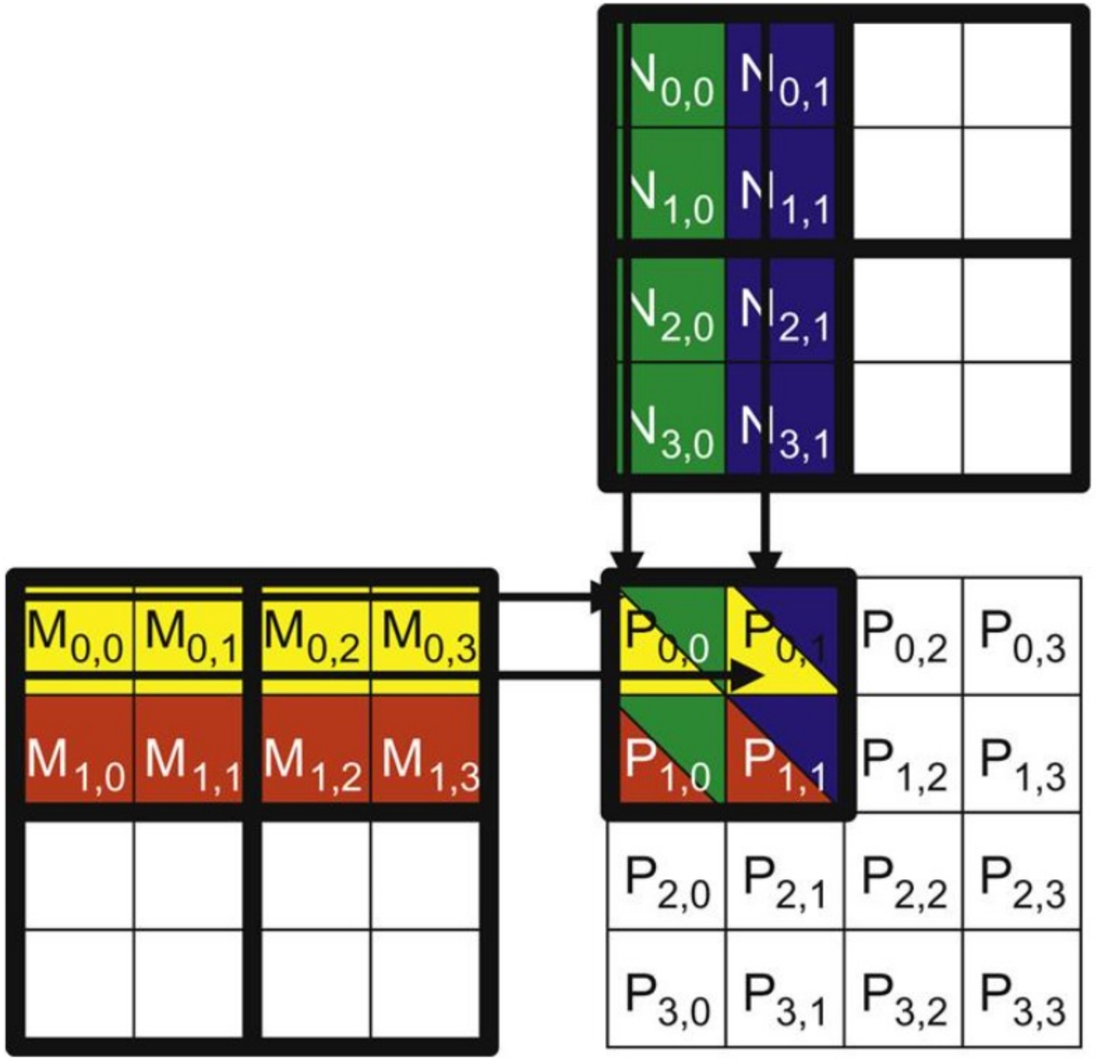
Tiling



Time step 1:

- Thread0,0 loads M0,0, N0,0.
- Thread0,1 loads M0,1, N0,1.
- Thread1,0 loads M1,0, N1,0.
- Thread1,1 loads M1,1, N1,1.

Tiling



Time step 2:

- Thread0,0 loads M0,2, N2,0.
- Thread0,1 loads M0,3, N2,1.
- Thread1,0 loads M1,2, N3,0.
- Thread1,1 loads M1,3, N3,1.

❖ Output P0,0, P0,1 need 2 time-steps to finish the computation

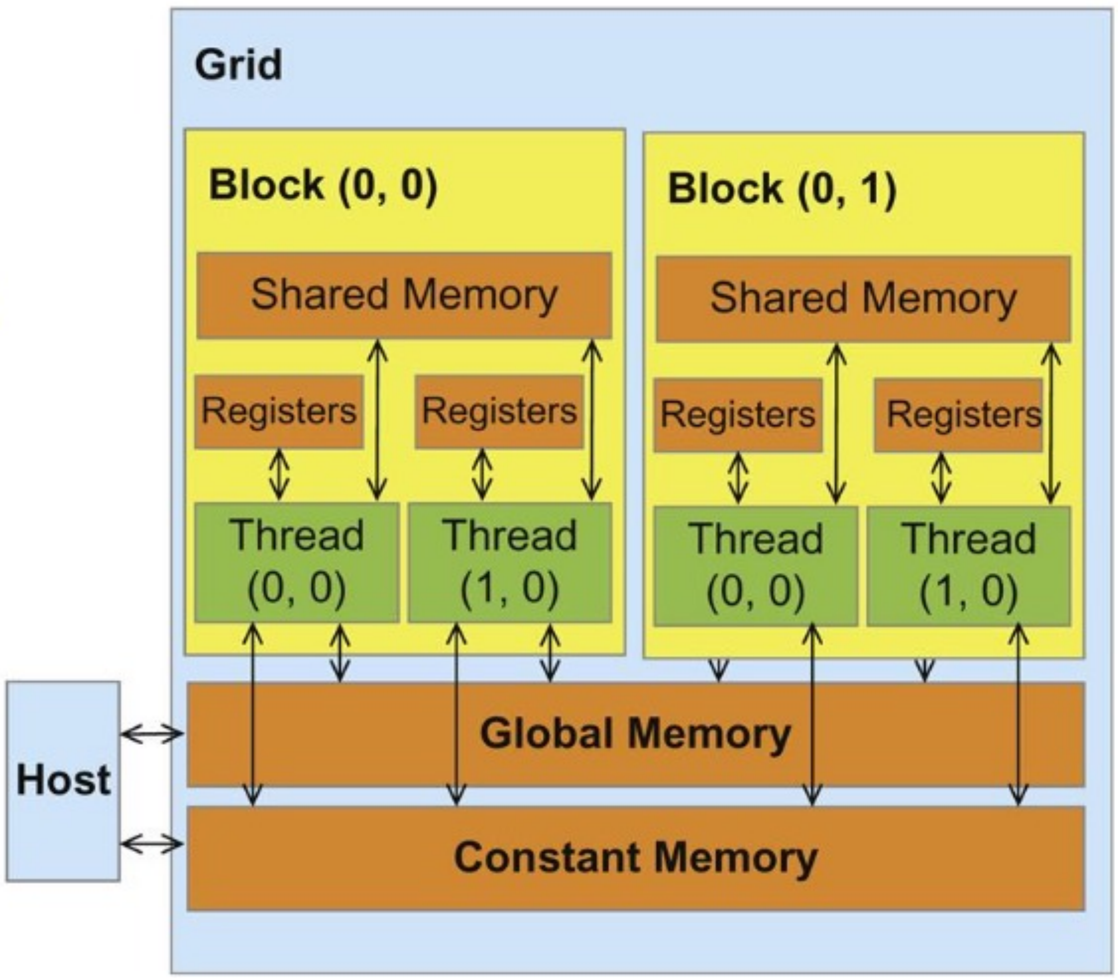
Tiled Version of Matrix Multiplication

```
#define TILE_WIDTH 2
__global__ void MatMulTiledKernel(float* d_M, float* d_N, float* d_P, int N) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Determine the row and col of the P element to be calculated for the thread
    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    for(int ph = 0; ph < N/TILE_WIDTH; ++ph) {
        Mds[ty][tx] = d_M[row * N + ph * TILE_WIDTH + tx];
        Nds[ty][tx] = d_N[(ph * TILE_WIDTH + ty) * N + col];
        __syncthreads();
        for(int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }
    d_P[row * N + col] = Pvalue;
}
```

compute-to-global-memory-access:
1.0 -> 4

Memory Restriction



If 1536 threads, 16384 registers

- Each thread can use only 16384/1536 = 10 registers

If 8 blocks, 16384 (16 K) bytes of shared memory

- Each block can use up to 2K of shared memory

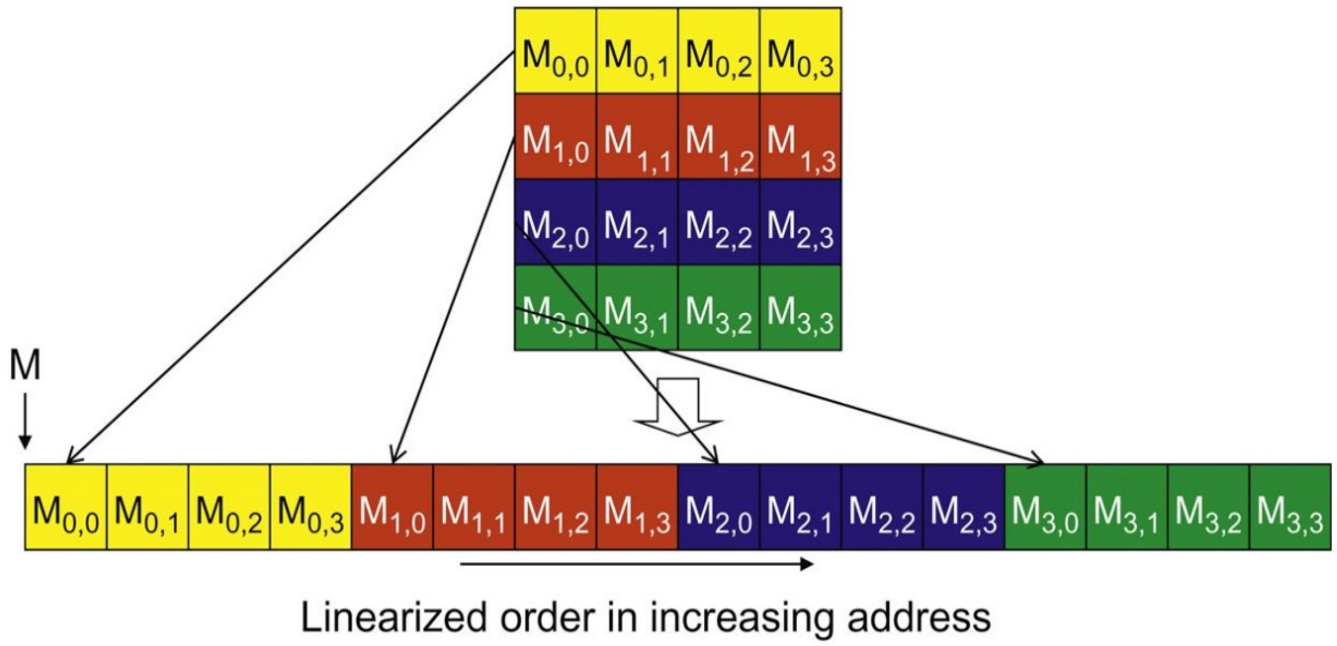
Tiled memory:

- Mds: 16 x 16 x 4 = 1KB
- Nds: 16 x 16 x 4 = 1KB

Today's Topic

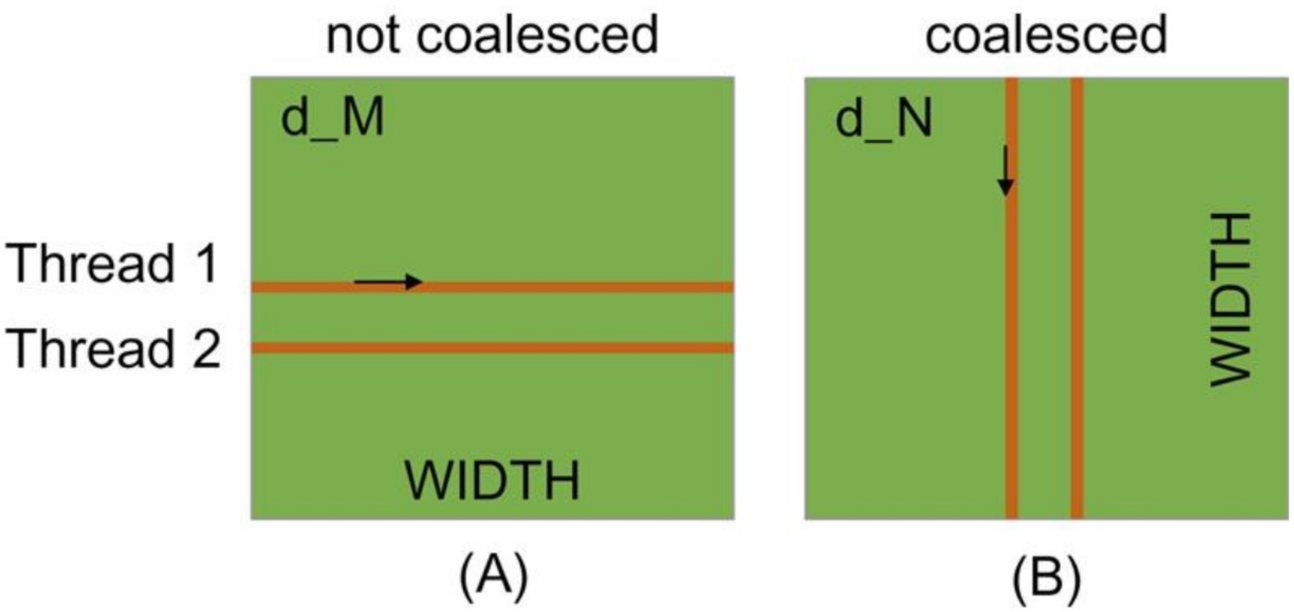
- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

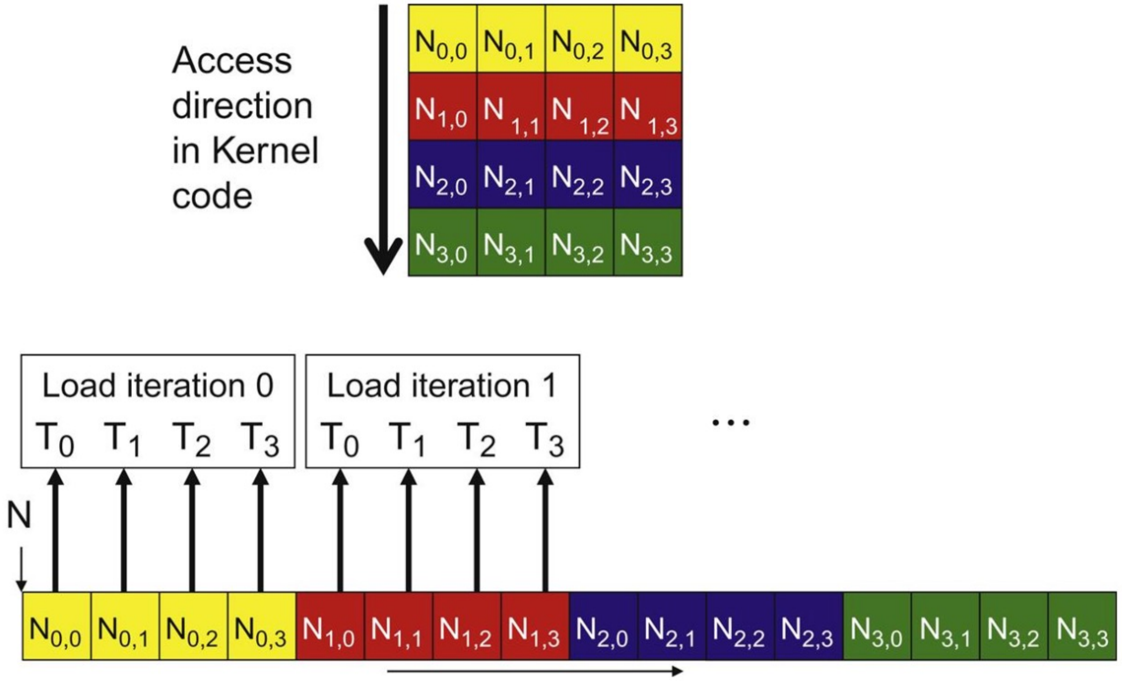
Locality / Bursts Organization



- Row-major format to store multidimensional array in C and CUDA
- The most favorable access pattern: when all threads in a warp access **consecutive** global memory locations
- The hardware combines, or coalesces, all these accesses
- These consecutive locations accessed and delivered are referred to as **DRAM bursts**

Coalesced Access





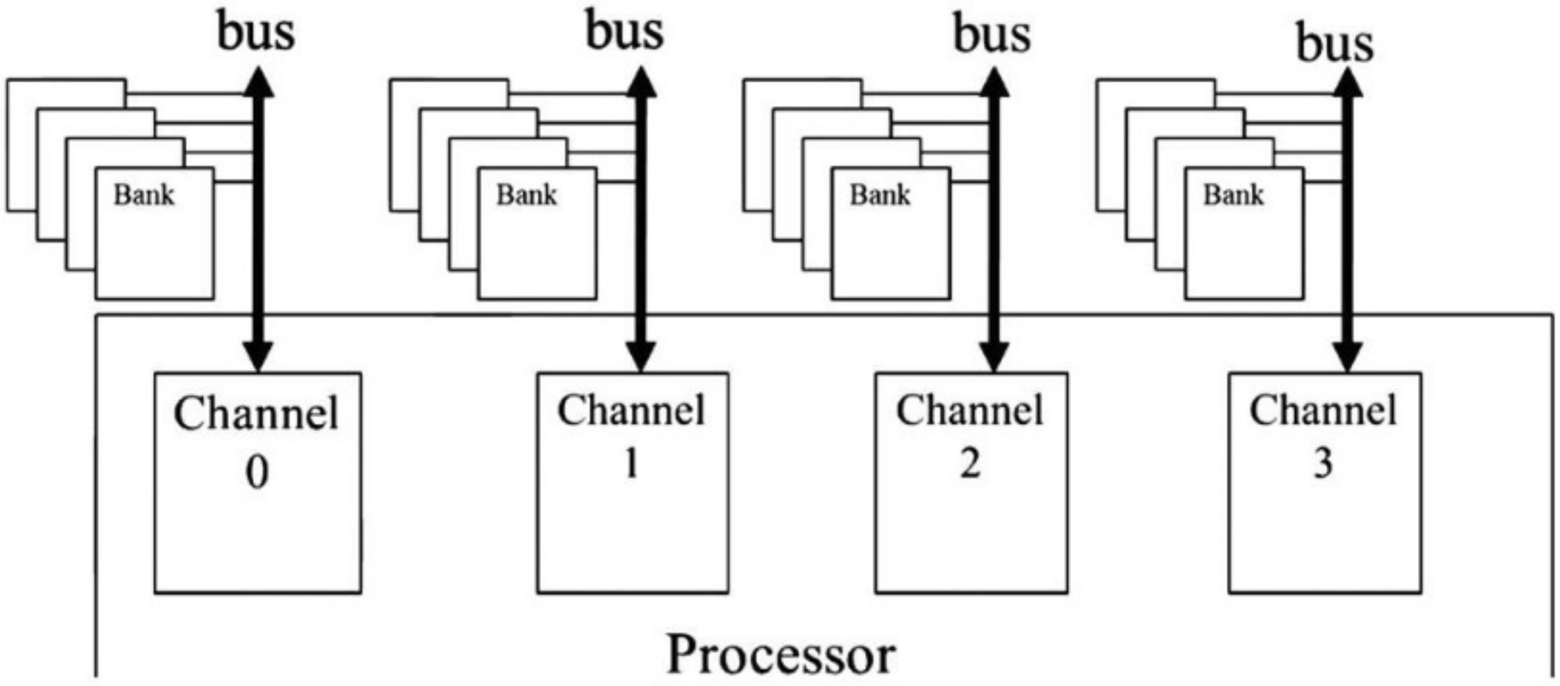
Recap of Matrix Multiplication

```
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
// Determine the row and col of the P element to be calculated for the thread
int row = by * TILE_WIDTH + ty;
int col = bx * TILE_WIDTH + tx;
float Pvalue = 0;
for(int ph = 0; ph < N/TILE_WIDTH; ++ph) {
    Mds[ty][tx] = d_M[row * N + ph * TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(ph * TILE_WIDTH + ty) * N + col];
    __syncthreads();
    for(int k = 0; k < TILE_WIDTH; ++k) {
        Pvalue += Mds[ty][k] * Nds[k][tx];
    }
    __syncthreads();
}
d_P[row * N + col] = Pvalue;
```

```
d_M[row][ph * TILE_WIDTH + tx]
d_N[ph * TILE_WIDTH + ty][col]
```

Each row of the tile is loaded by `TILE_WIDTH` threads whose `threadIdx` are identical in the y dimension and **consecutive** in the x dimension.

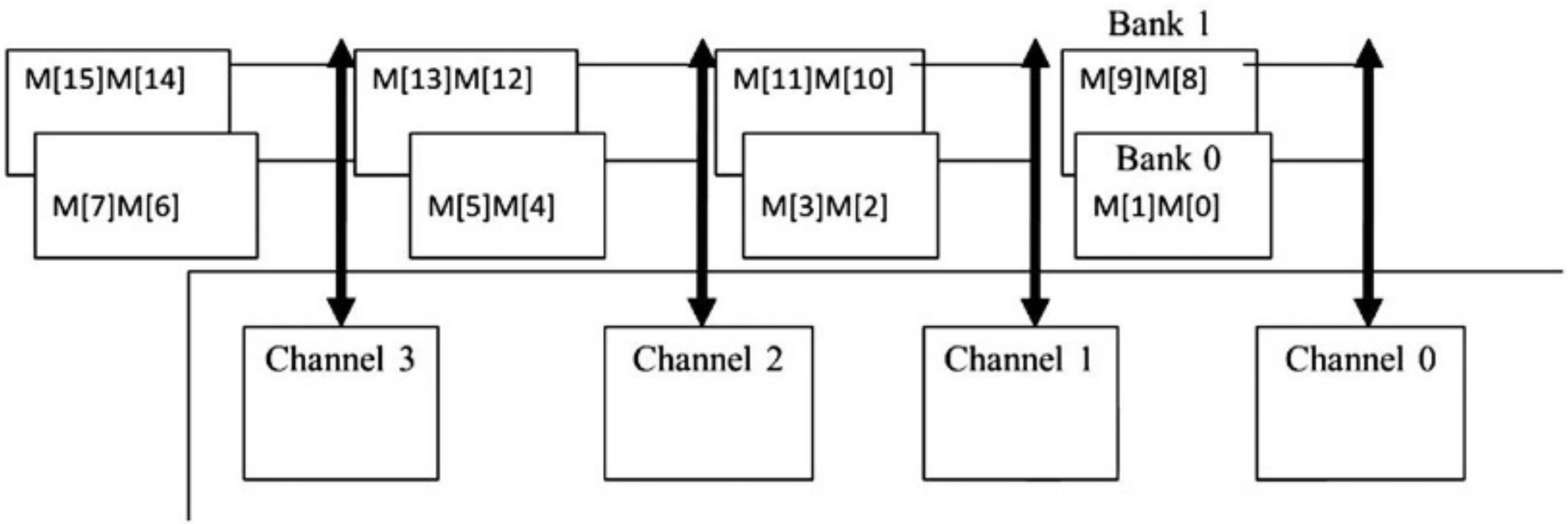
Memory Parallelism



Banks and channels

- a processor that contains four channels, each with a bus that connects four DRAM banks to the processor
- Limited by data transfer bandwidth of the bus -> connecting multiple banks to a channel bus

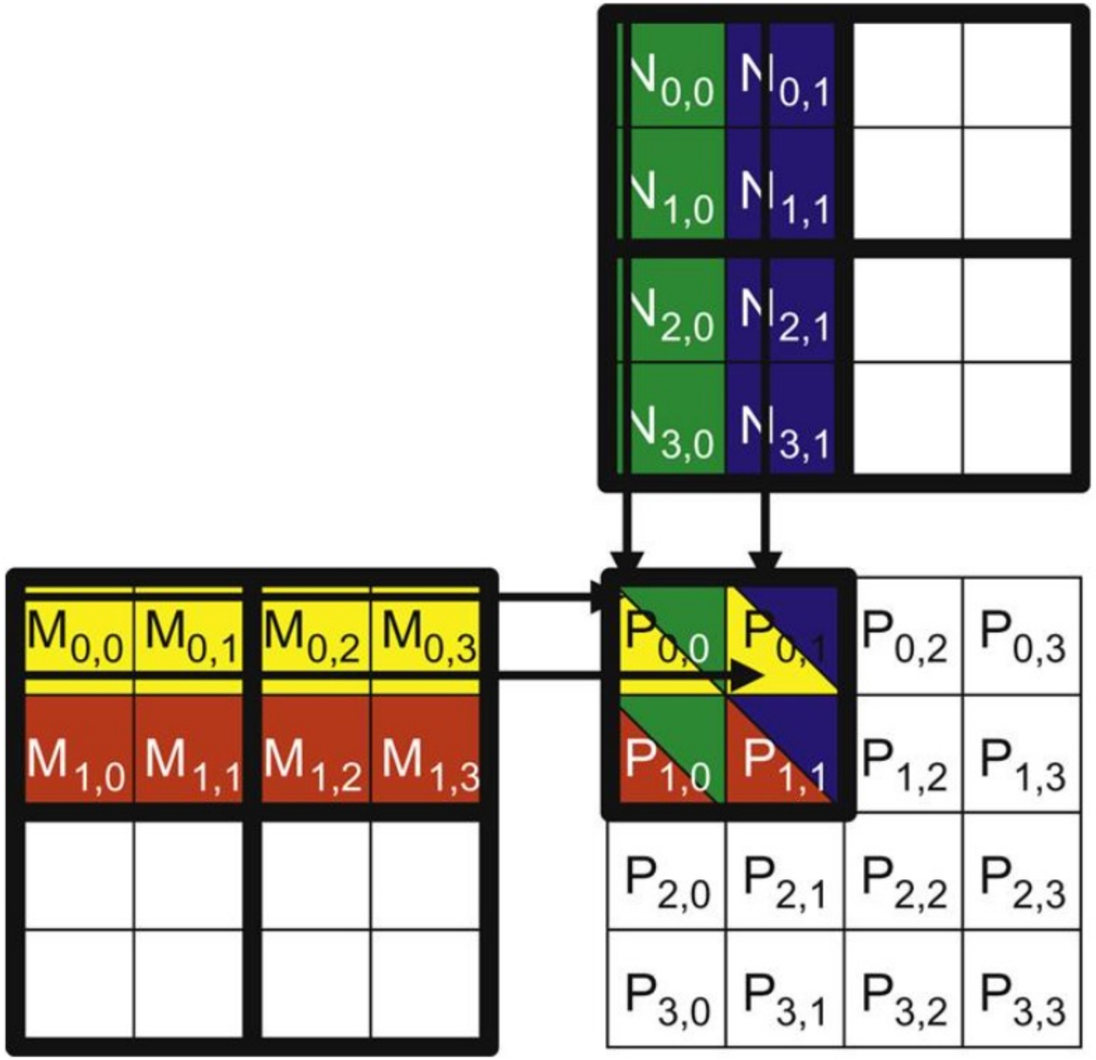
Interleaved Data Distribution



Banks and channels

- This scheme ensures that even relatively small arrays are spread out nicely.
- Assume a small burst size of two elements (8 bytes)

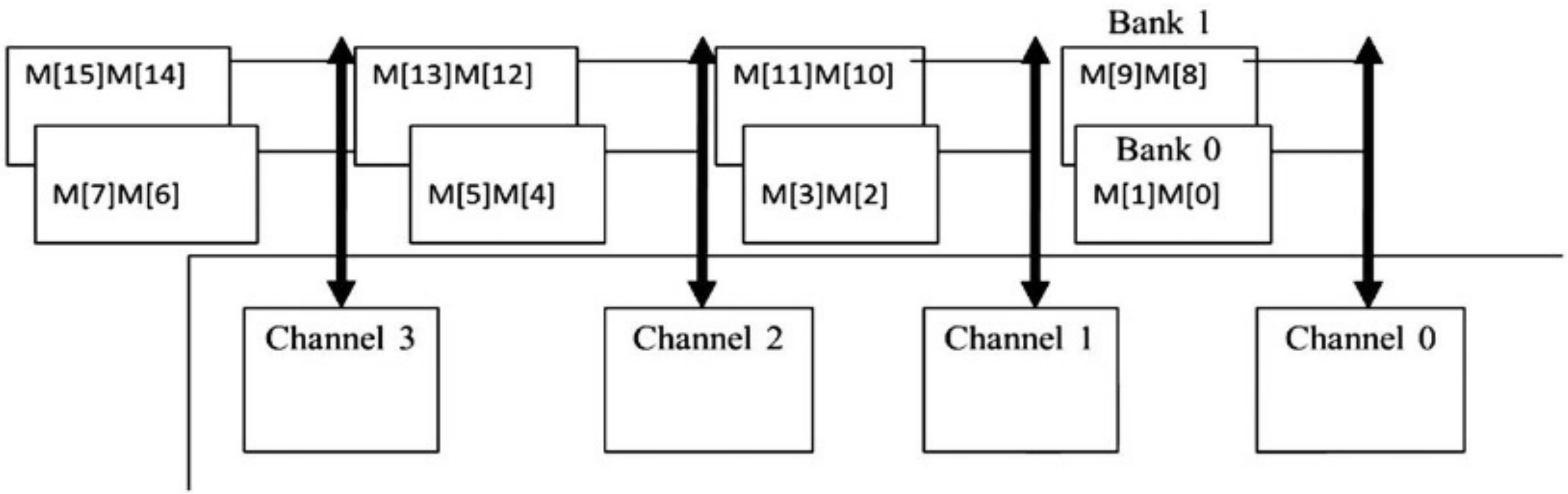
Memory Parallelism



Assume a 2x2 thread blocks and 2x2 tiles.

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase0	M[0][0], M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase1	M[0][2], M[0][3], M[1][2], M[1][3]	M[0][2], M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]

Memory Parallelism

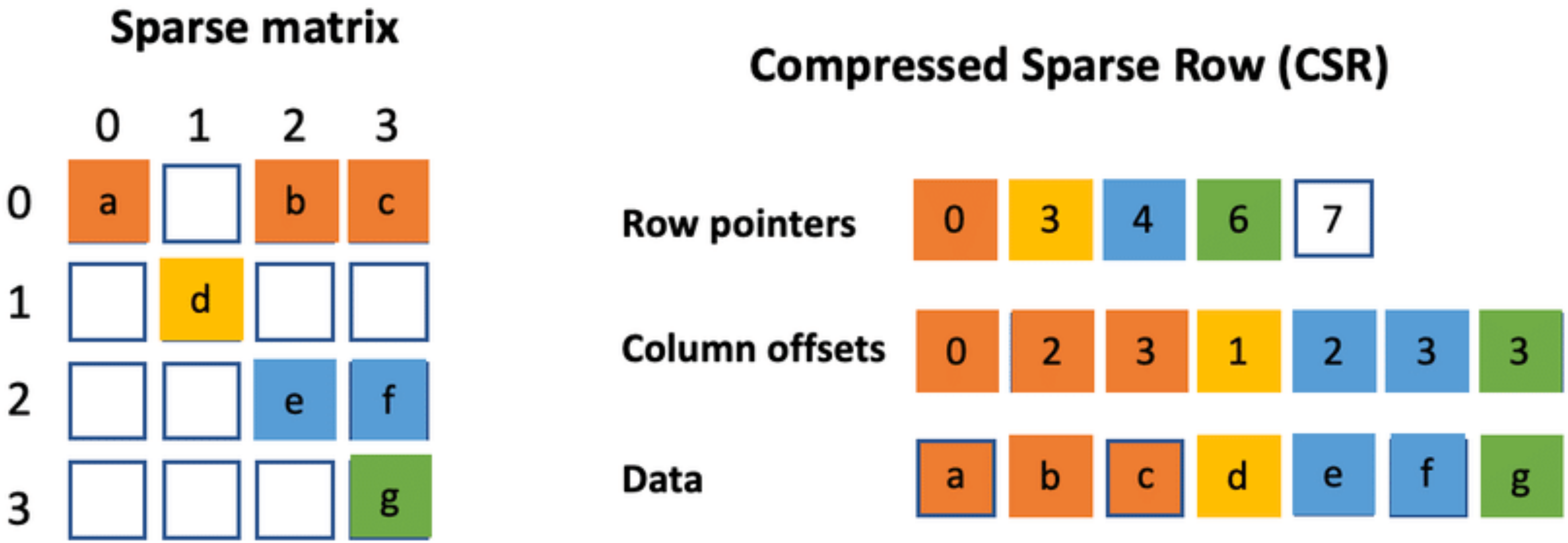


Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase0	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase1	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

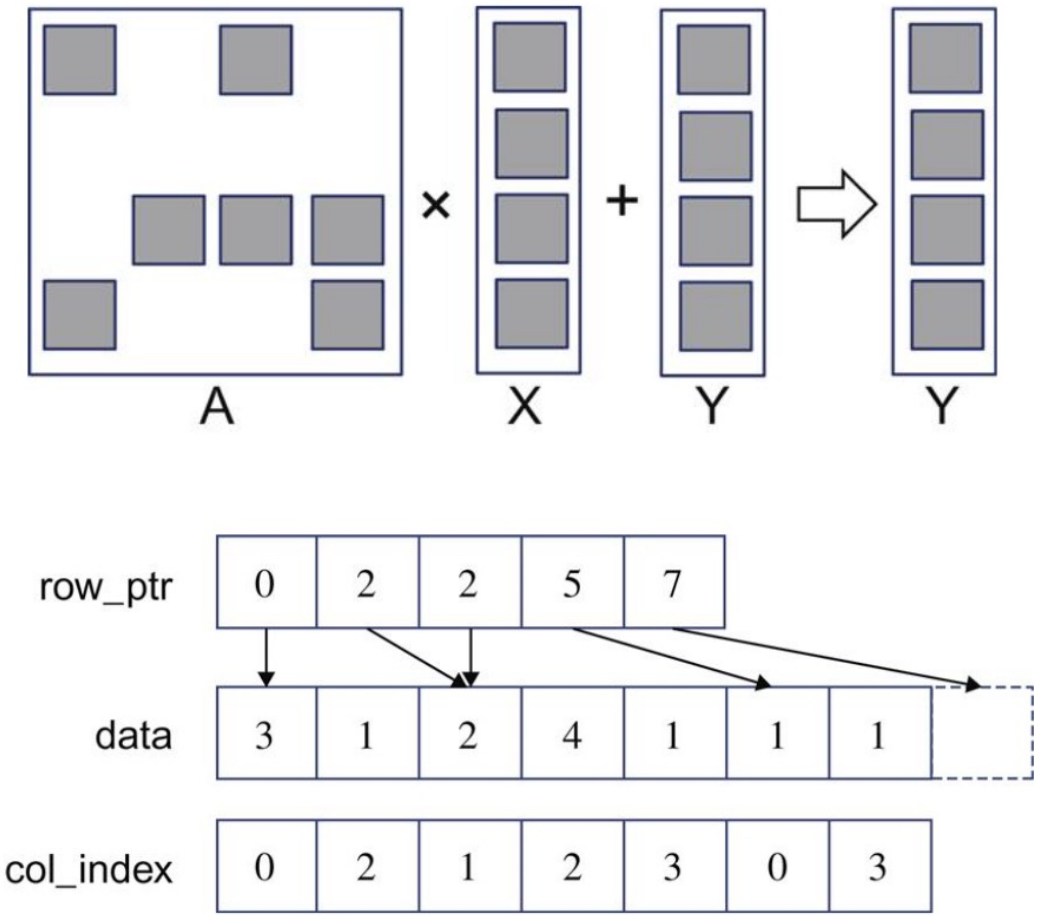
Today’s Topic

- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

Sparse Matrix - CSR



Sparse Matrix-Vector Multiplication




```
for(int row = 0; row < n; row++) {
    float dot = 0;
    int row_start = row_ptr[row];
    int row_end = row_ptr[row + 1];
    for(int el = row_start; el < row_end; el++)
    {
        dot += x[el] * data[col_index[el]];
    }
    y[row] += dot;
}
```

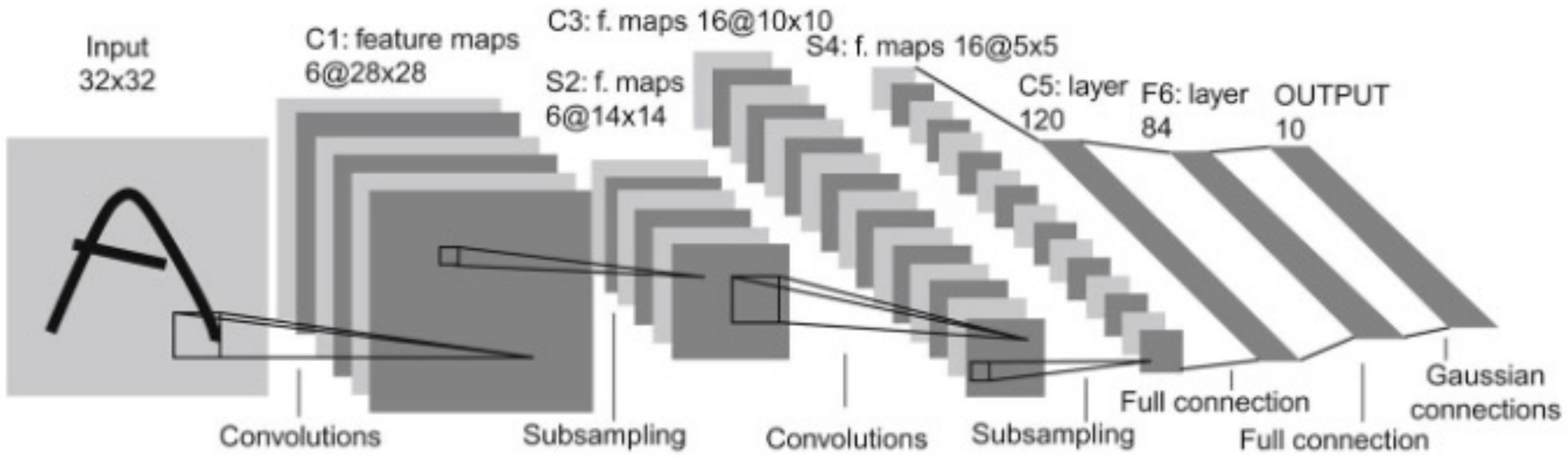
Sparse Matrix-Vector Multiplication

```
__global__ void SpMVCSRKernel(float *data, int *col_index, int
*row_ptr, float *x, float *y, int num_rows) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if(row < num_rows) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row + 1];
        for(int elem = row_start; elem < row_end; elem++) {
            dot += x[row] * data[col_index[elem]];
        }
        y[row] += dot;
    }
}
```

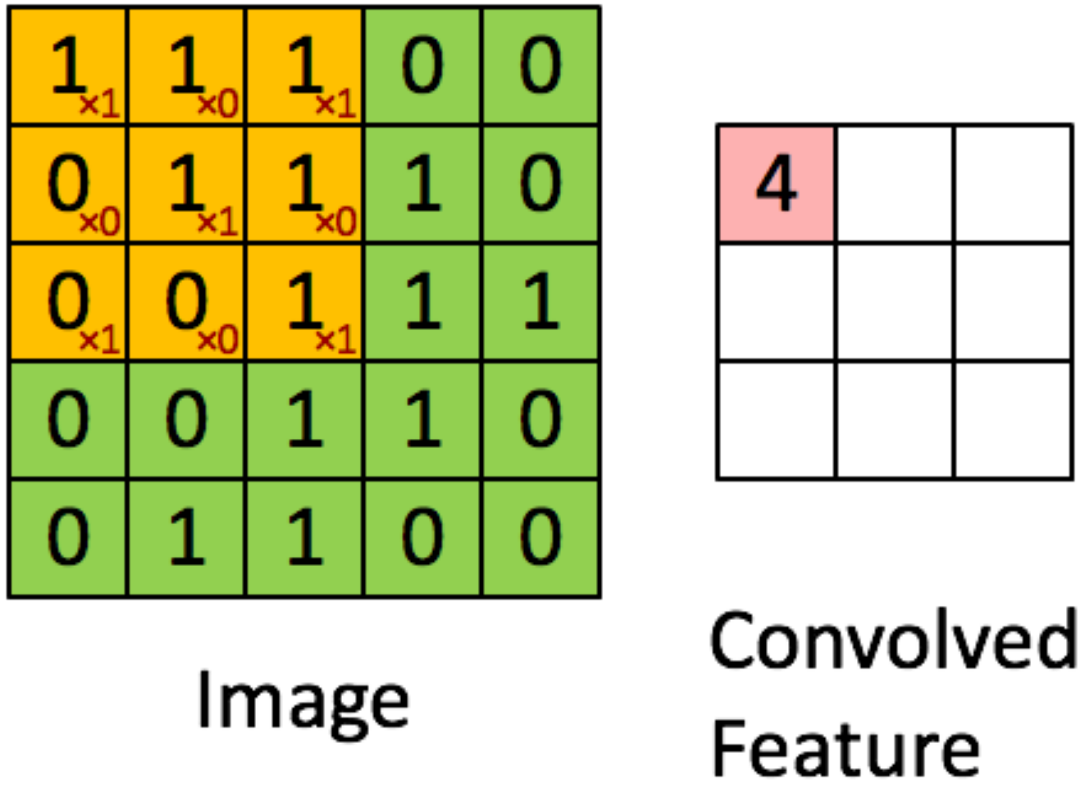
Today's Topic

- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

Convolutional Neural Network



Convolutional Layer



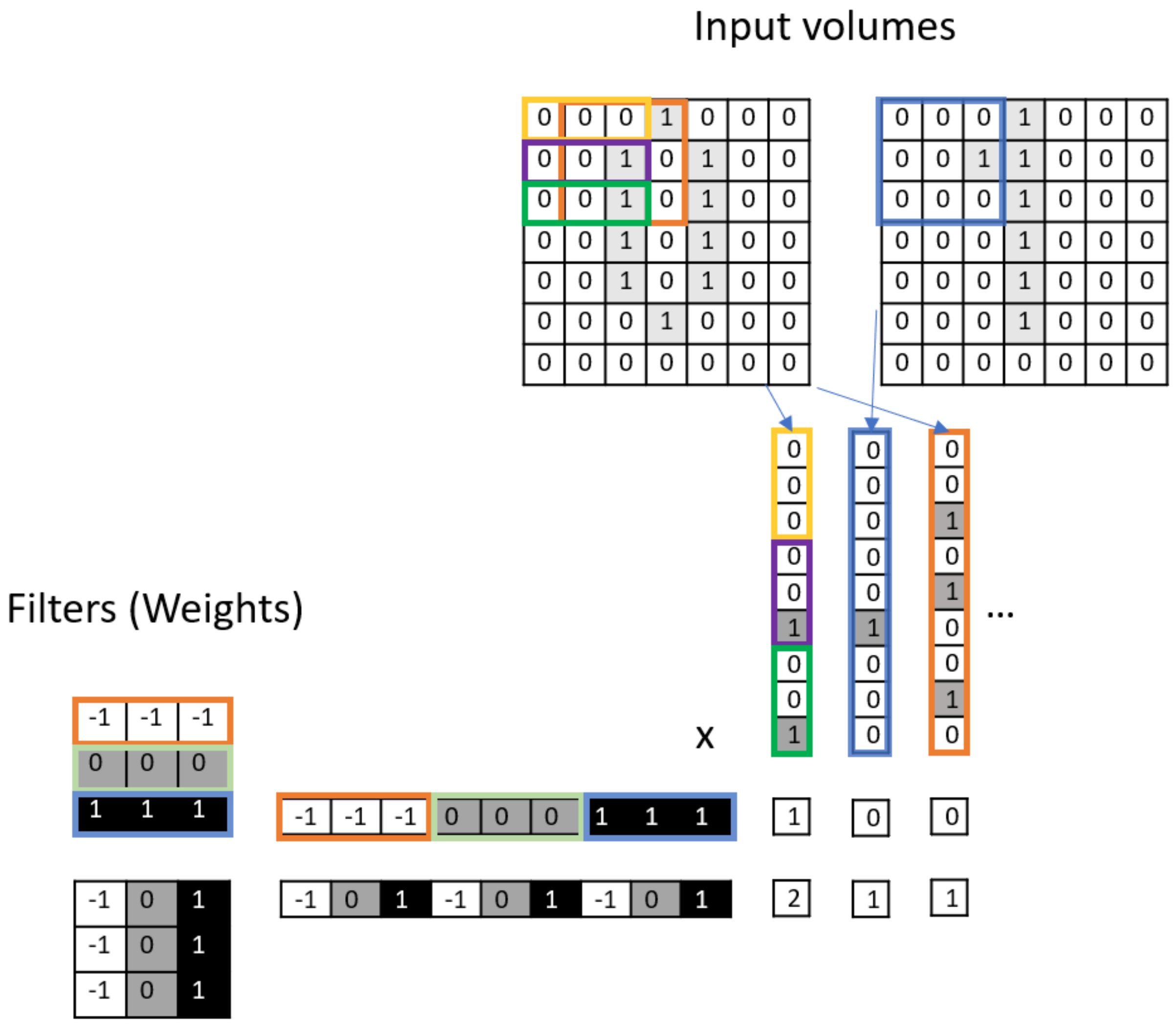

```
// kernel: C_OUT * C_IN * K * K
// input: N * C_IN * H * W
// output: N * C_OUT * h_out * w_out
```

```
void naive_conv(int N, int H, int W, int K, int C_IN,
int C_OUT, float *input, float *output, float *kernel) {
    int h_out = H - K + 1;
    int w_out = W - K + 1;
    for(int n = 0; n < N; n++)
        for(int c_in = 0; c_in < C_IN; c_in++)
            for(int c_out = 0; c_out < C_OUT; c_out++)
                for(int h = 0; h < h_out; h++)
                    for(int w = 0; w < w_out; w++)
                        for(int i = 0; k < K; i++)
                            for(int j = 0; j < K; j++)

}
```

```
output[n,c_out,h,w] +=
input[n,c_in,h+i,w+j]* kernel[c_out,c_in,i,j];
```

Convolution as Matrix Multiplication



- Filters:

C_out x C_in x K x K -> [C_out] [C_in x K^2]

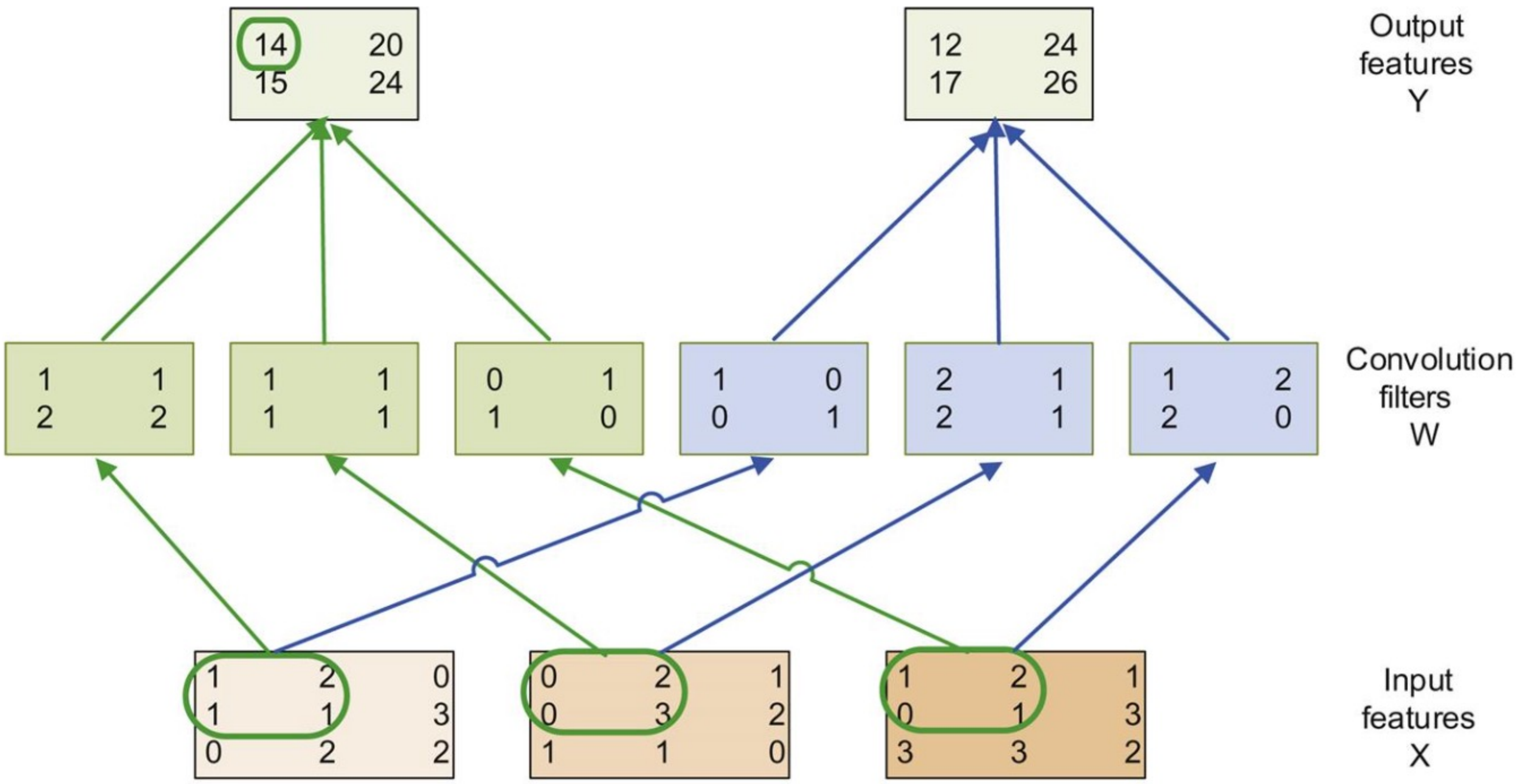
- Input Matrix:

N x C_in x H x W -> [N] [C_in x K^2] [H_out x W_out]

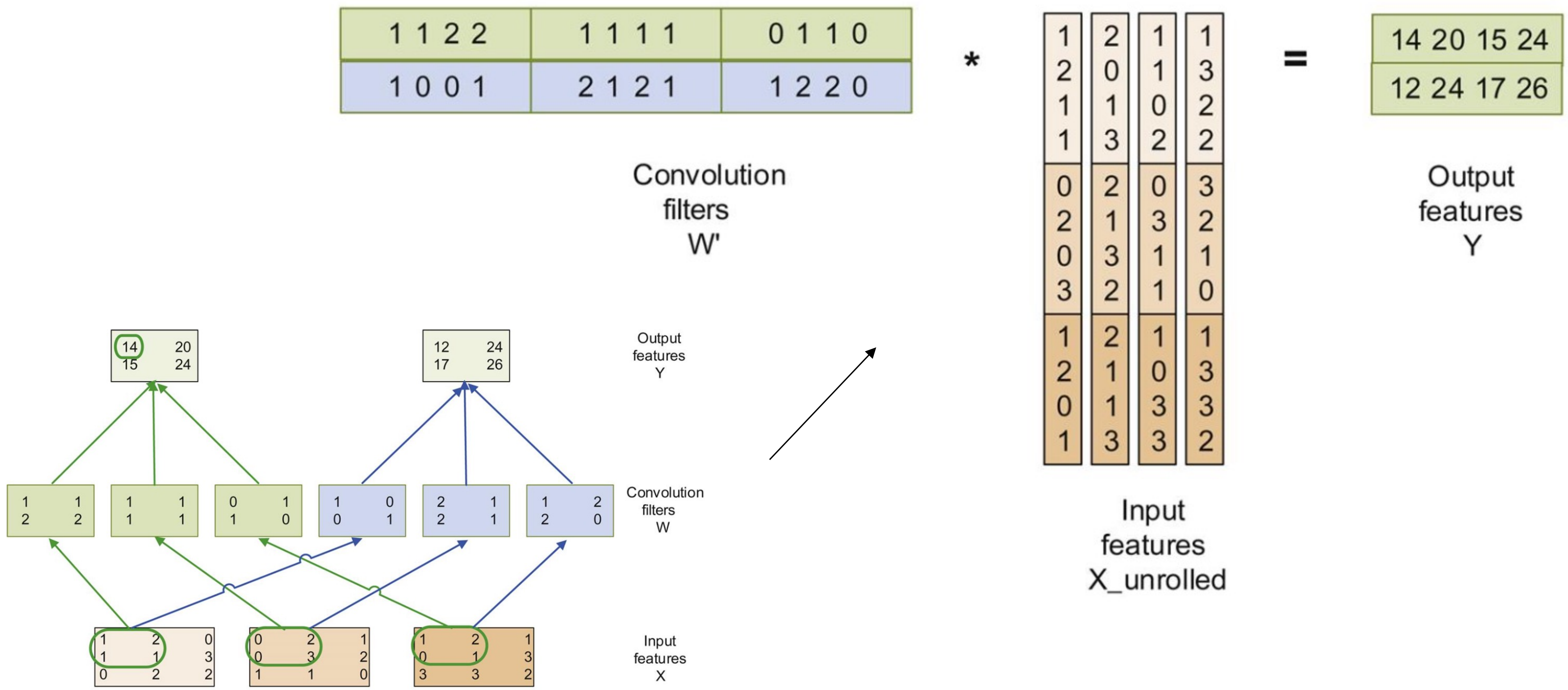
- Output Matrix:

N x C_out x H_out x W_out

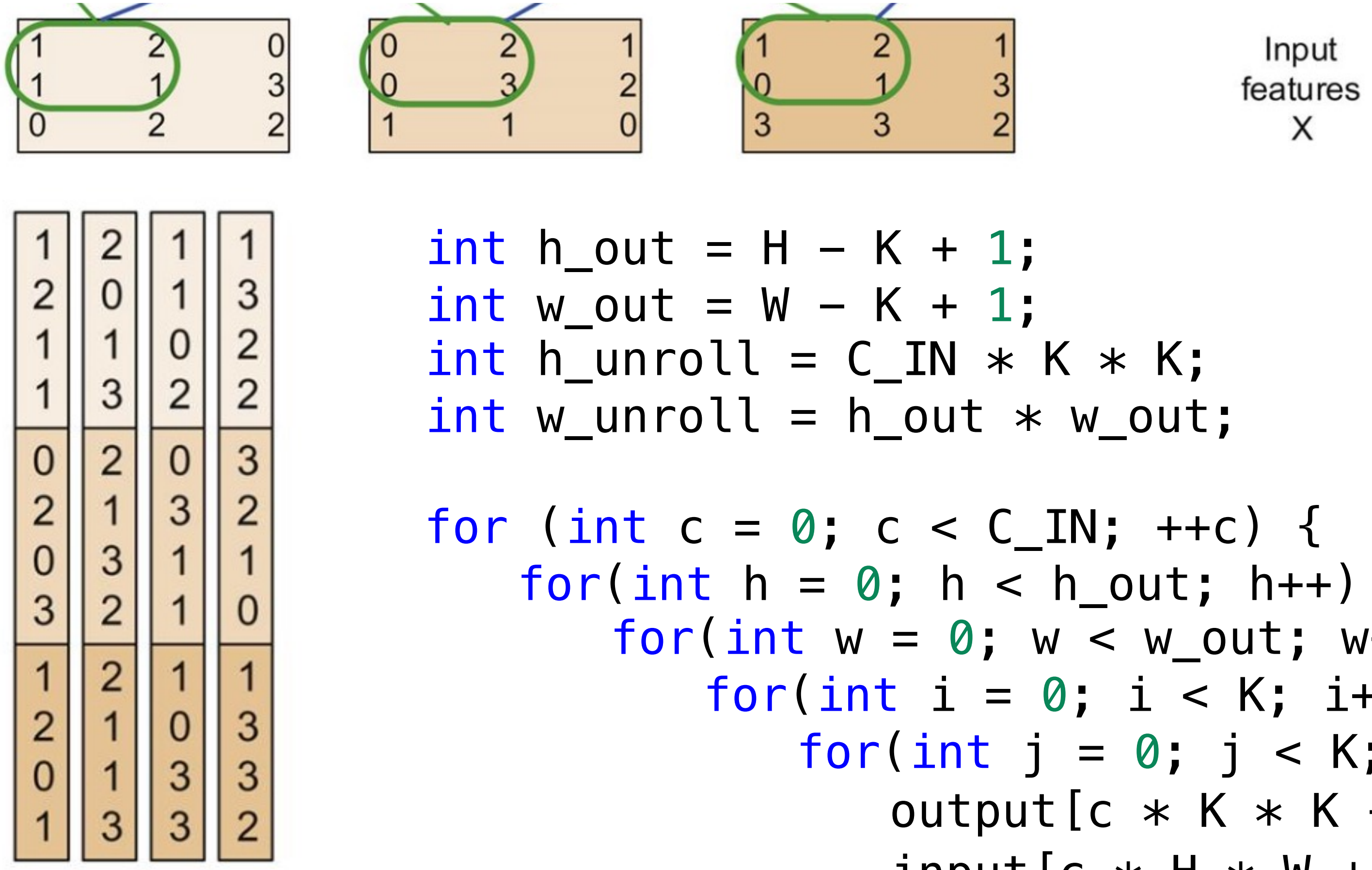
Convolution as Matrix Multiplication



Convolution as Matrix Multiplication



Im2col: Unroll the Input Matrix




```
int h_out = H − K + 1;
int w_out = W − K + 1;
int h_unroll = C_IN ∗ K ∗ K;
int w_unroll = h_out ∗ w_out;

for (int c = 0; c < C_IN; ++c) {
    for(int h = 0; h < h_out; h++) {
        for(int w = 0; w < w_out; w++) {
            for(int i = 0; i < K; i++) {
                for(int j = 0; j < K; j++) {
                    output[c ∗ K ∗ K + h ∗ w_out + w][i ∗ K + j] =
                    input[c ∗ H ∗ W + (h + i) ∗ W + w + j]; }}}}
}
```

Today's Topic

- Tiling (Chap 4)
- Memory parallelism (Chap 4 & 5)
- Accelerating Matrix Multiplication on GPU (Chap 4 & 5)
- Sparse Matrix (Chap 10.2)
- Convolution as Matrix Multiplication (Chap 16.4)
- cuBLAS

Reading for Next Class

LightSeq: A High Performance Inference Library for Transformers. Wang et al. NAACL 2021.

LightSeq2: Accelerated Training for Transformer-based Models on GPUs. Wang et al. SC 2022.