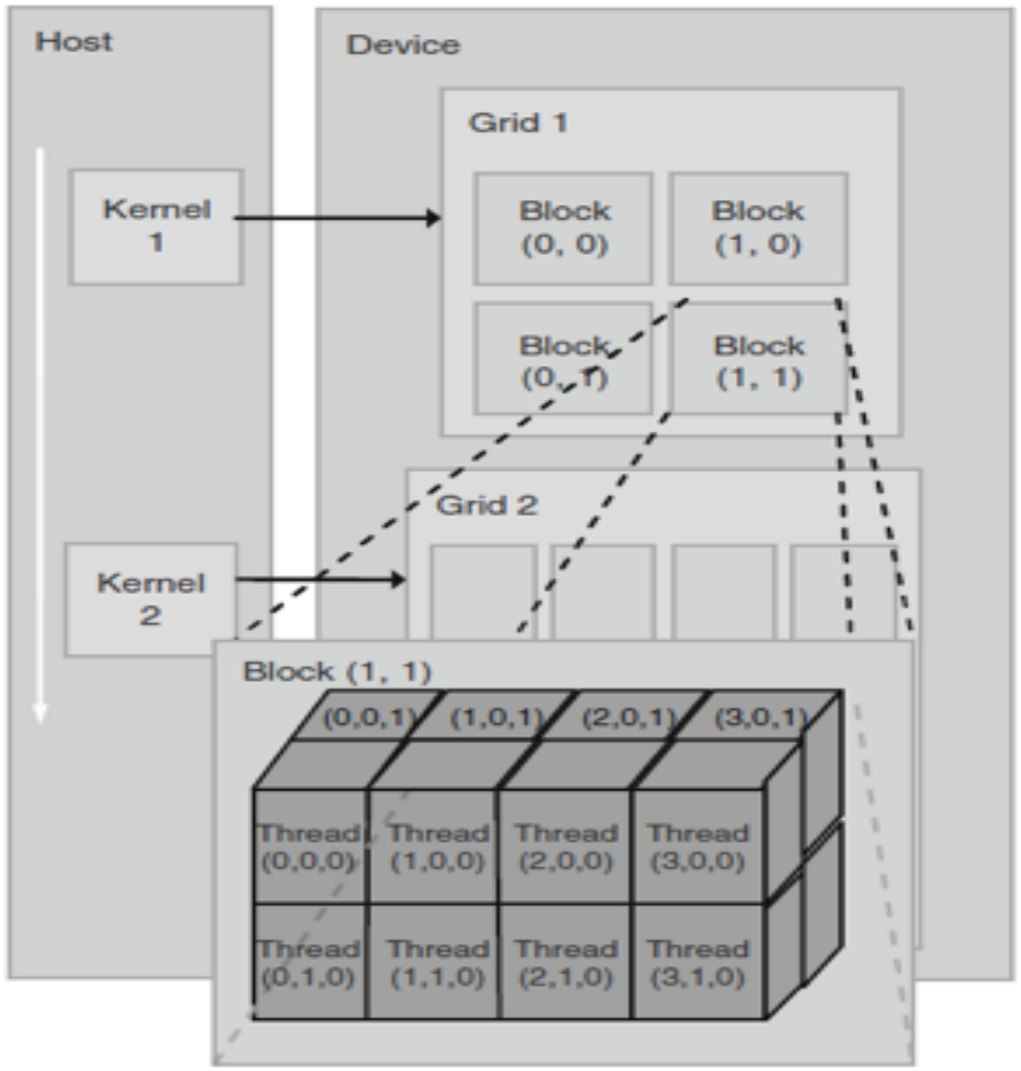
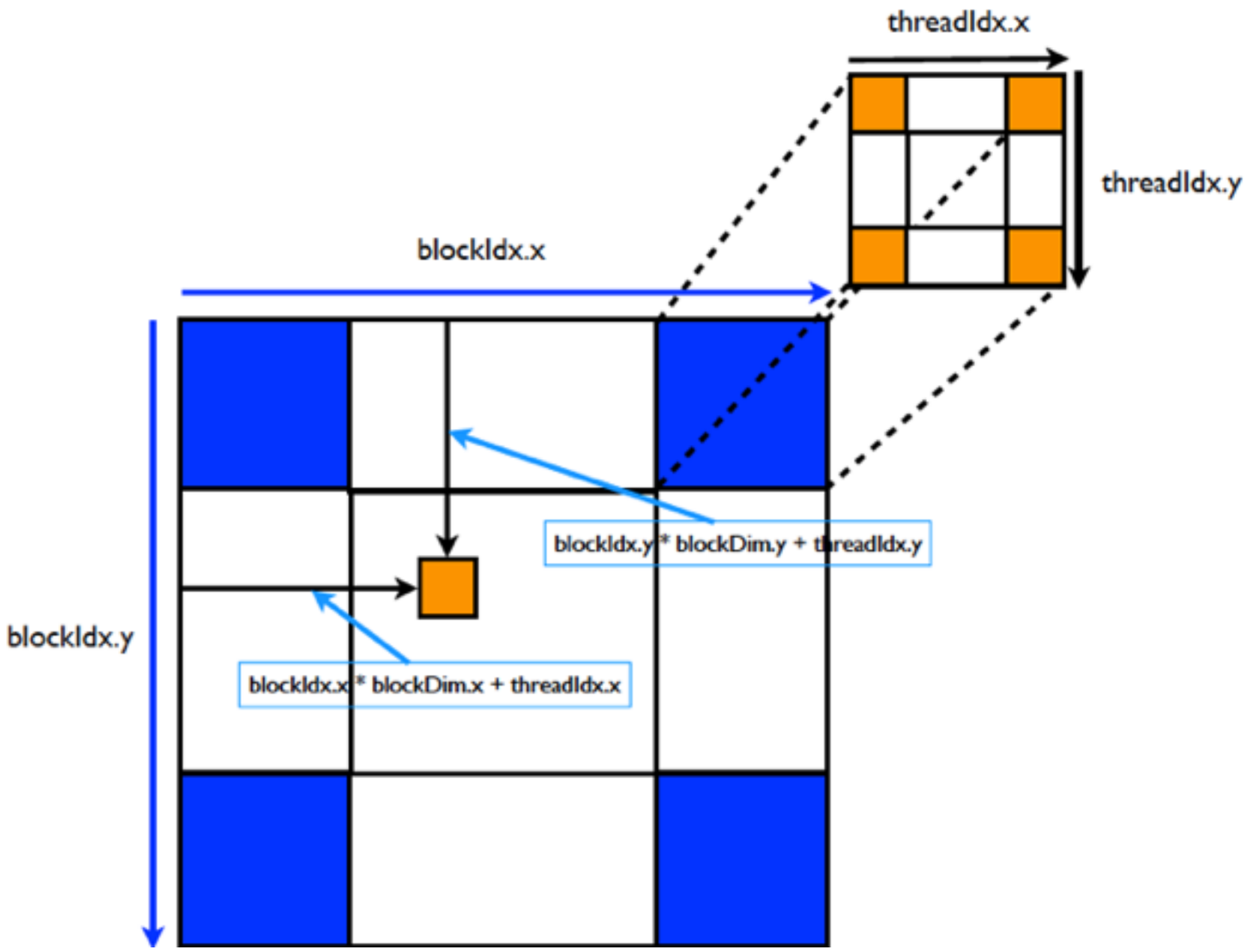
Data parallel Execution Model

CUDA THREAD ORGANIZATION

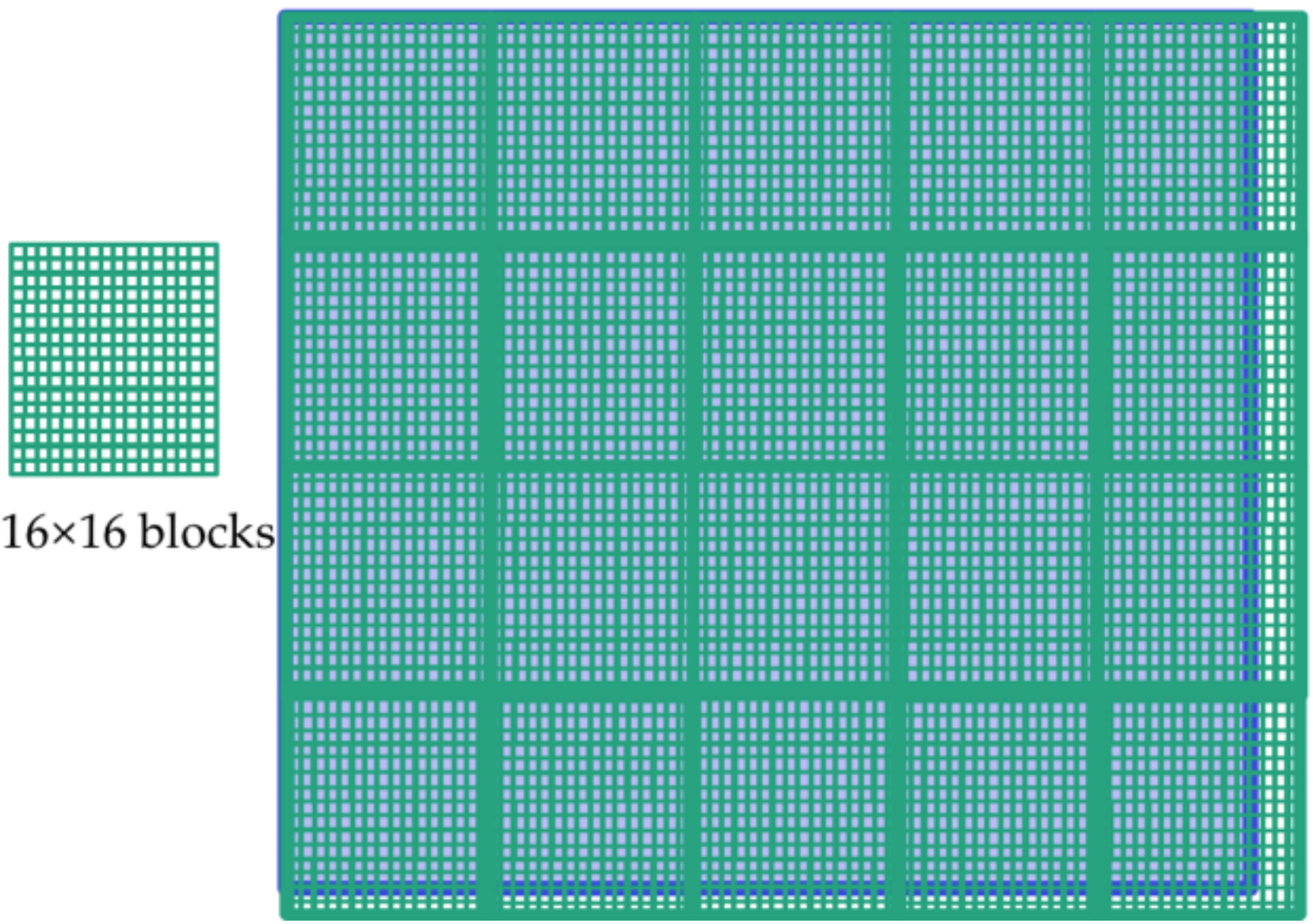


- dim3 blockDim(2, 2, 1);
- dim3 gridDim(4, 2,2);
- kernelFunction<<< gridDim, blockDim>>>(...);
- Each block is labeled with (blockIdx.x, blockIdx.y) ex: block(1,0)-> blockIdx.x=1 and blockIdx.y=0
- ThredIdx also consists of 3 fields: threadIdx.x, threadIdx.y, threadIdx.z. Ex: thread (3, 1,0) ->threadIx.x=3, threadIdx.y=1, threadIdx.z=0
- Total number of threads= 4*16=64

Mapping Threads to Multidimensional Data

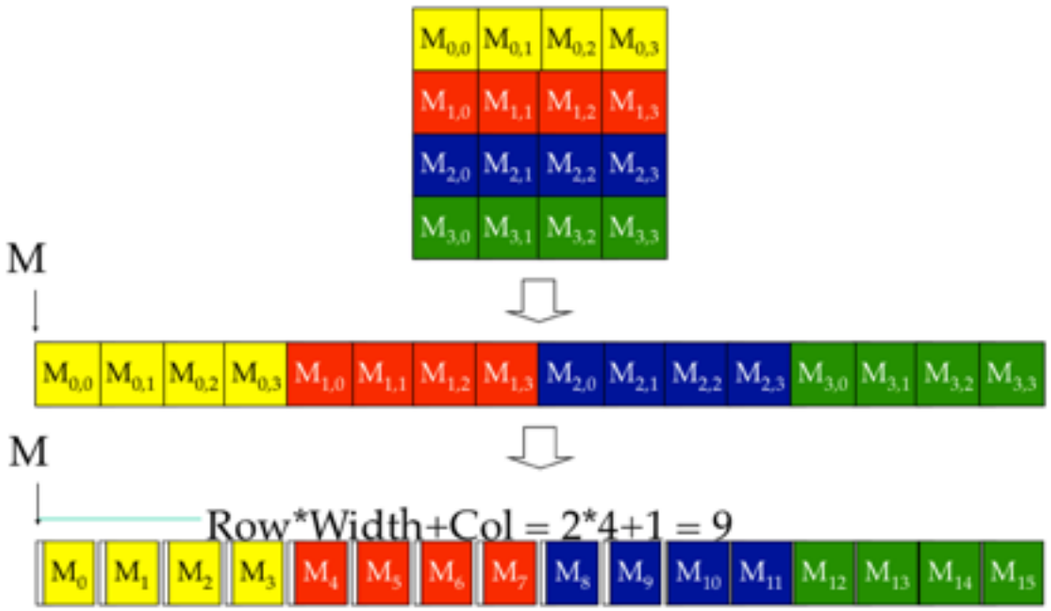


Mapping Threads to Multidimensional Data ...



- 76 X 62 picture(76 in horizontal direction and 62 pixels in vertical direction)
- Suppose we use 16 x 16 block with 16 threads in x direction and 16 threads in y direction.
- 5 -> direction 4-> y direction (80 x 64)

Mapping Threads to Multidimensional Data ...



- **Multi-dimensional arrays are linearized**
- **Row-Major layout**
  - **Place all elements of the same row into consecutive locations**
  - **Rows are then placed one after another into the memory space.**

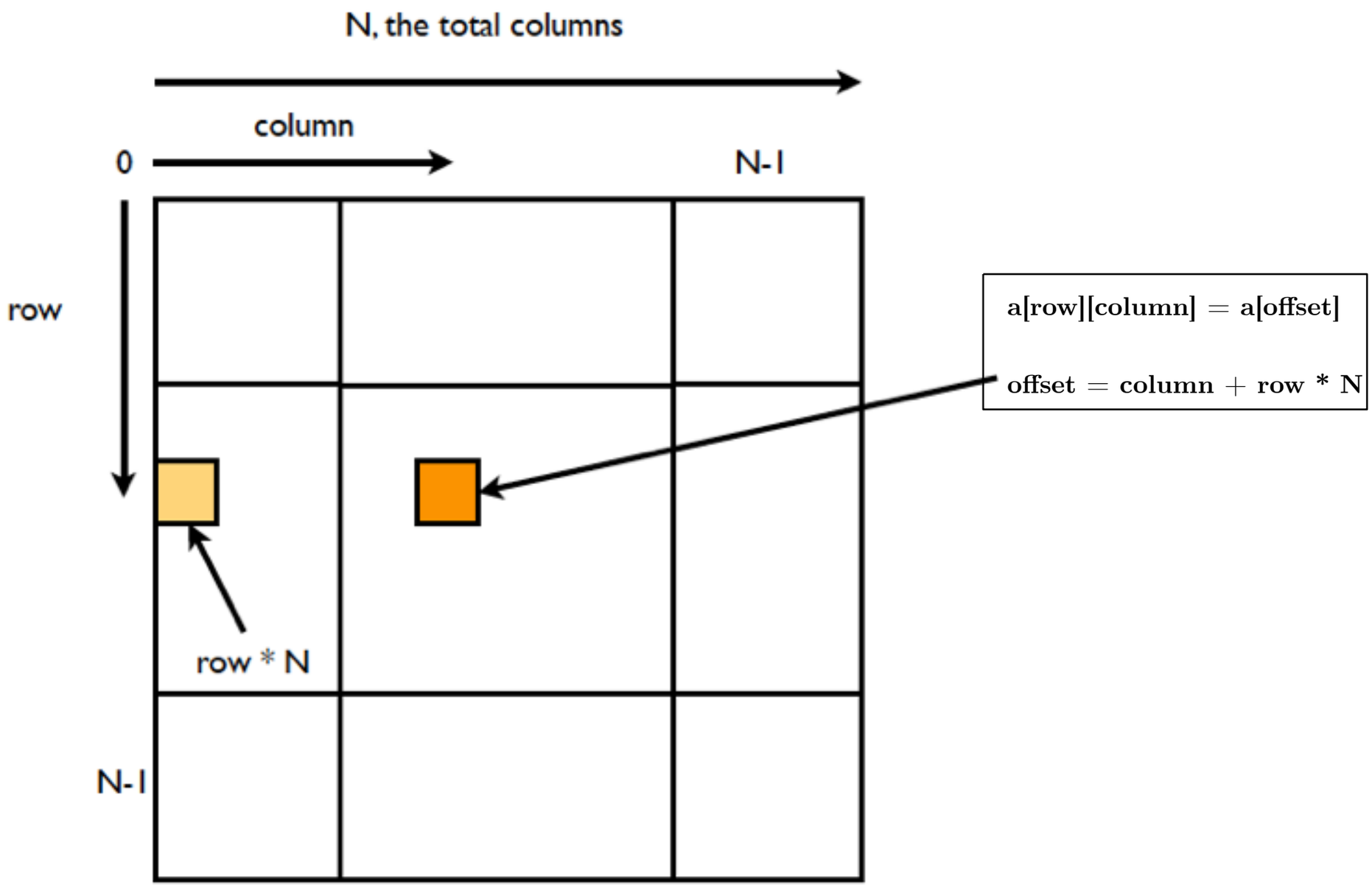
1D index for M element in row i and column j is i*4+j, where

i*4 skips over all elements of the rows before row i

Term j selects the right element within the section for row i

Index for $M_{2,1}$ is 2*4+1 = 9 = M_9

Mapping Threads to Multidimensional Data ...



2D image Scaling Kernel

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int m,int n)
{

  // Calculate the row # of the d_Pin and d_Pout element to process
  int row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element to proce
  int col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    d_Pout[row*n+col] = 2*d_Pin[row*n+col];
  }
}
```

Assignment

- ✦ Read an image pixels into 2D array in C and launch picture scaling kernel.

Matrix Addition Example

$$c_{ij} = a_{ij} + b_{ij} \ (0 <= i < m,\ 0 <= j < n)$$

Matrix Addition Example...

```
#define N 512
#define BLOCK_DIM 512

__global__ void matrixAdd (int *a, int *b, int *c);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

int size = N * N * sizeof(int);

// initialize a and b with real values (NOT SHOWN)

cudaMalloc((void**)&dev_a, size);
cudaMalloc((void**)&dev_b, size);
cudaMalloc((void**)&dev_c, size);

cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
```

Matrix Addition Example...

```
 dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
 dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));

 matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}
```

```
__global__ void matrixAdd (int *a, int *b, int *c) {
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 int row = blockIdx.y * blockDim.y + threadIdx.y;

 int index = col + row * N;

 if (col < N && row < N) {
  c[index] = a[index] + b[index];
 }
}
```

Querying Devices

Need to allocate memory and execute code on device

Useful to know how much memory and the capabilities of the device

cudaGetDeviceCount() : get count of CUDA devices

Devices capable of executing kernels written in CUDA C

```
int count;
cudaGetDeviceCount(&count);
```

After getting the count, we can iterate through the devices and query relevant information about each device.

CUDA runtime returns us these properties in a structure of type *cudaDeviceProp*.

Possible Device Properties

```
struct cudaDeviceProp {
    char name[256];                  int kernelExecTimeoutEnabled;
    size_t totalGlobalMem;           int integrated;
    size_t sharedMemPerBlock;        int canMapHostMemory;
    int regsPerBlock;                int computeMode;
    int warpSize;                    int maxTexture1D;
    size_t memPitch;                 int maxTexture2D[2];
    int maxThreadsPerBlock;          int maxTexture3D[3];
    int maxThreadsDim[3];            int maxTexture2DArray[3];
    int maxGridSize[3];              int concurrentKernels;
    size_t totalConstMem;    }
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
```

Possible Device Properties...

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
`int computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited

Program to Read Device Properties

```
int main()
{
    // Number of CUDA devices
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);
    // Iterate through devices
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }
    printf("\nPress any key to exit...");
    char c;
    scanf("%c", &c);
    return 0;
}
```

Program to Read Device Properties...

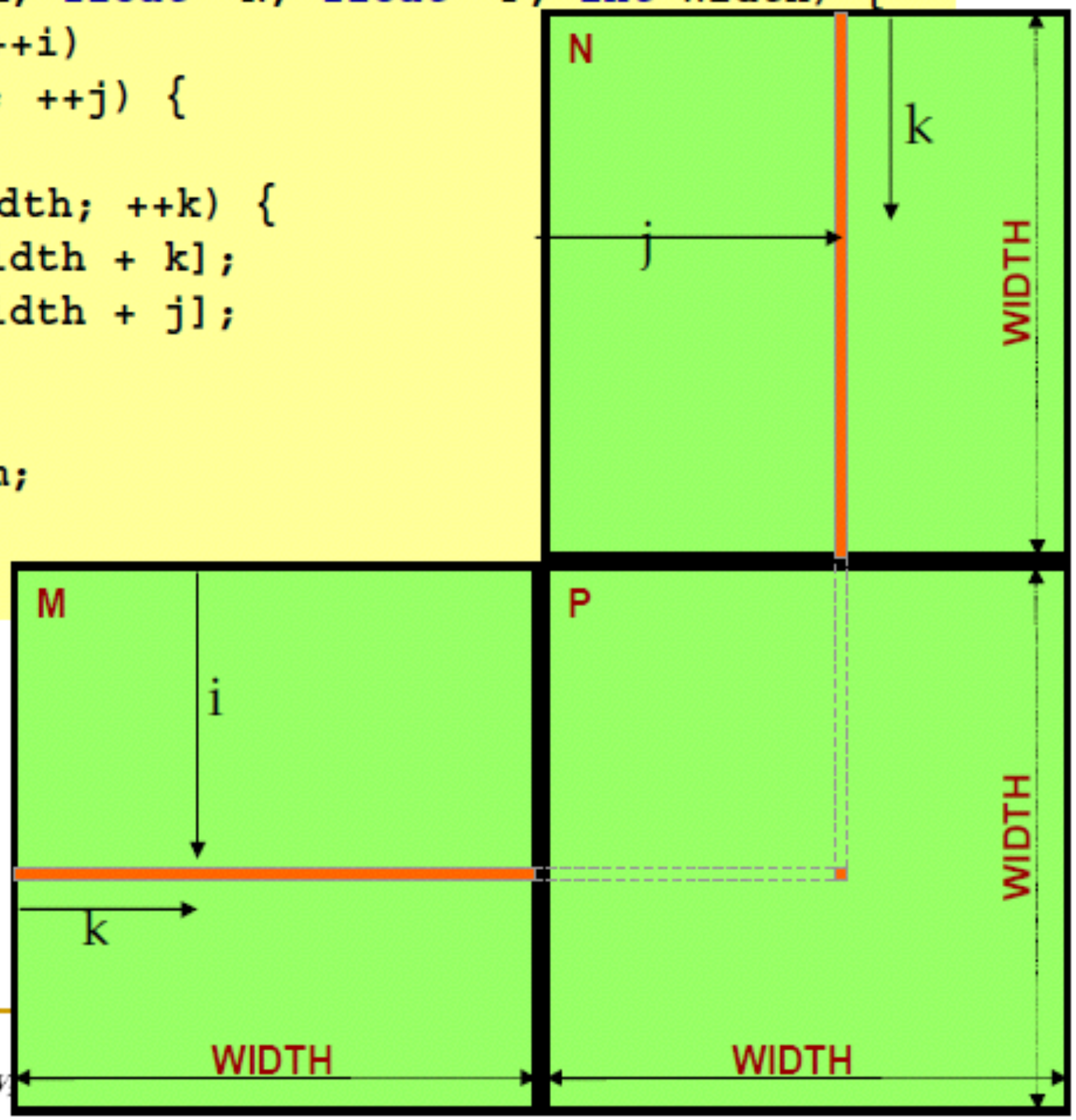
```
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number:         %d\n",  devProp.major);
    printf("Minor revision number:         %d\n",  devProp.minor);
    printf("Name:                          %s\n",  devProp.name);
    printf("Total global memory:           %u\n",  devProp.totalGlobalMem);
    printf("Total shared memory per block: %u\n",  devProp.sharedMemPerBlock);
    printf("Total registers per block:     %d\n",  devProp.regsPerBlock);
    printf("Warp size:                     %d\n",  devProp.warpSize);
    printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
    printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
    printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
    printf("Clock rate:                    %d\n",  devProp.clockRate);
    printf("Total constant memory:         %u\n",  devProp.totalConstMem);
    printf("Texture alignment:             %u\n",  devProp.textureAlignment);
    printf("Concurrent copy &execution: %s\n",  (devProp.deviceOverlap ? "Yes" : "No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout: %s\n",  (devProp.kernelExecTimeoutEnabled ? "Yes" : "No"));
        return;
}
```

Output on TeslaK20x Card

```
[user2@diamond testCuda]$ nvcc -o deviceQuery deviceQuery.cu
[user2@diamond testCuda]$ ./deviceQuery
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K20Xm
Total global memory:           1744371712
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    732000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     14
Kernel execution timeout:      No

Press any key to exit...
```

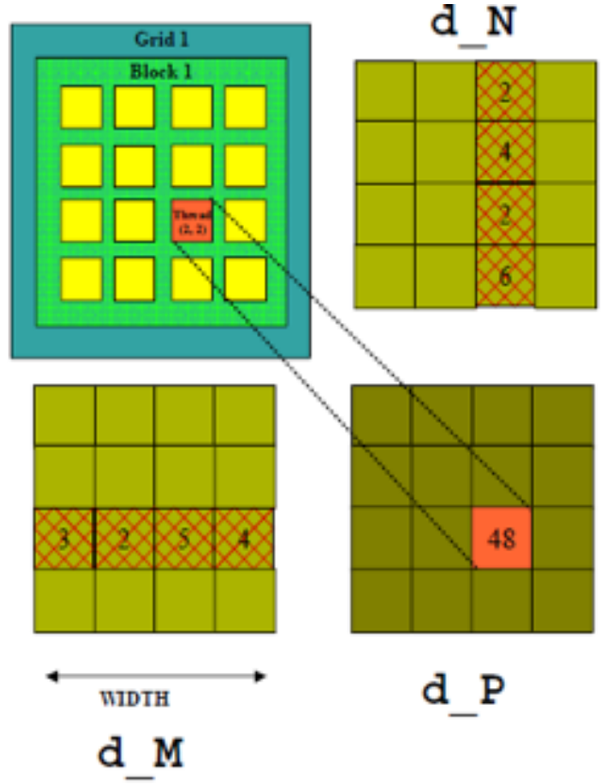
Matrix Multiplication -Serial Code

```
void MatrixMulOnHost( float* M, float* N, float* P, int Width) {
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
        float sum = 0;
        for (int k = 0; k < Width; ++k) {
             float a = M[i * Width + k];
             float b = N[k * Width + j];
             sum += a * b;
        }
        P[i * Width + j] = sum;
    }
}
```



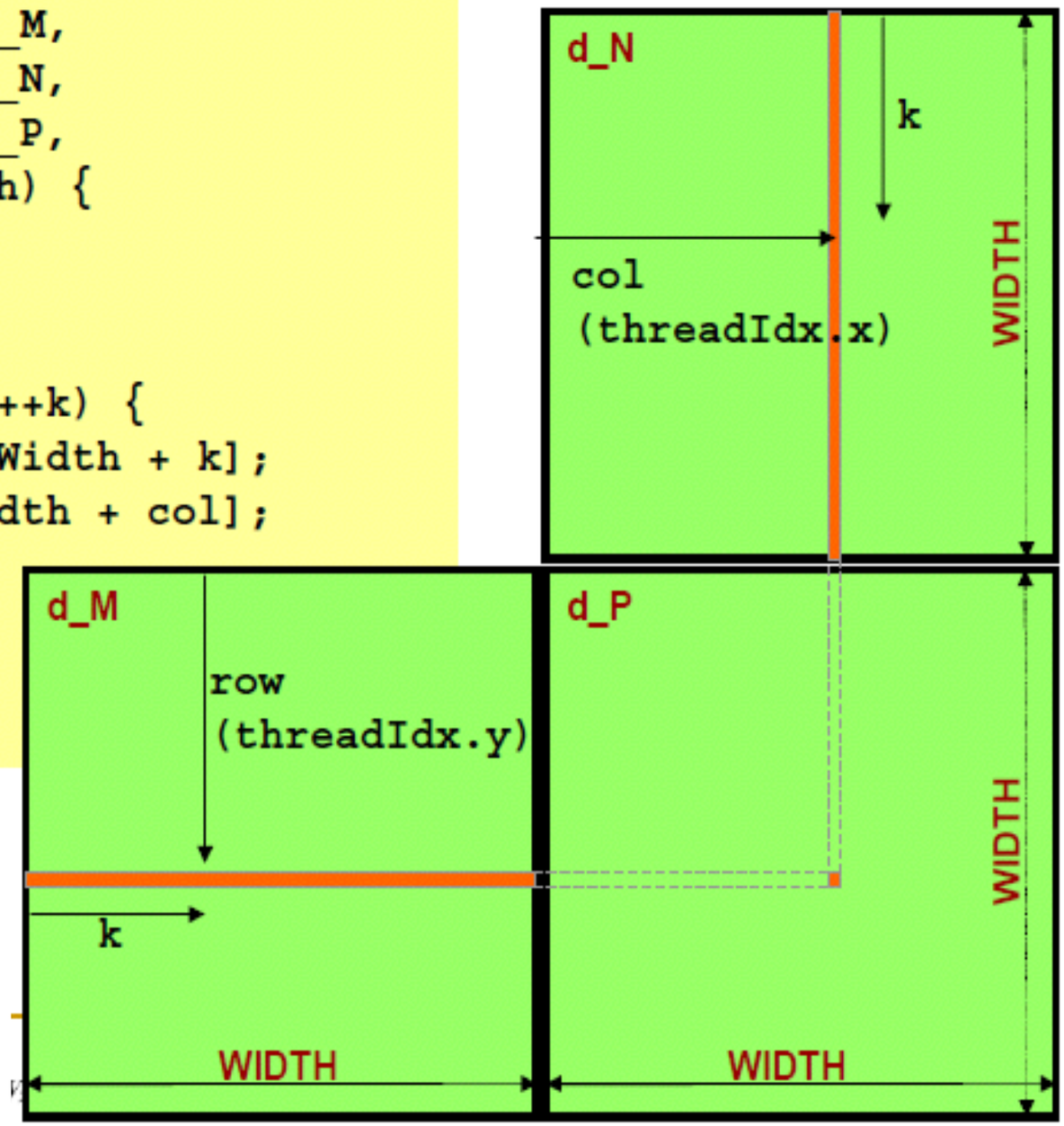
Matrix Multiplication -Parallel Code

- **Single** Block of threads compute matrix d_P
- Each thread
  - Loads a row of the matrix d_M
  - Loads a column of the matrix d_N
  - Perform one multiply and addition for each pair of d_M and d_N elements
  - Computes one element of d_P



Matrix Multiplication -Parallel Code...

```
__global__
void MatrixMulKernel(float* d_M,
                     float* d_N,
                     float* d_P,
                     int Width) {
  int row = threadIdx.y;
  int col = threadIdx.x;
  float P_val = 0;
  for (int k = 0; k < Width; ++k) {
    float M_elem = d_M[row * Width + k];
    float N_elem = d_N[k * Width + col];
    P_val += M_elem * N_elem;
  }
  d_p[row*Width+col] = P_val;
}
```



Matrix Multiplication(MM) -Host Code

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width){
    int matrix_size = Width * Width * sizeof(float);
    float *d_M, *d_N, *d_P;

    //Allocate and Load M and N to device memory
    cudaMalloc((void **) &d_M, matrix_size);
    cudaMemcpy(d_M, M, matrix_size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_N, matrix_size);
    cudaMemcpy(d_N, N, matrix_size, cudaMemcpyHostToDevice);

    //Allocate P on the device
    cudaMalloc((void **) &d_P, matrix_size);
```

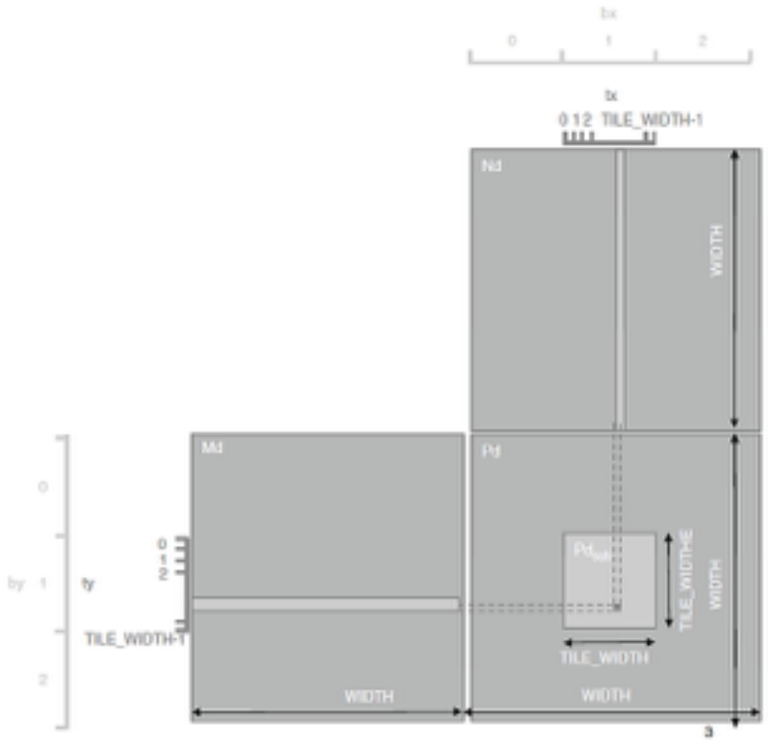
Matrix Multiplication -Host Code...

```
//setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);


//Launch the device computation threads
MatrixMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N,  d_P, Width);

        //Copy back the results from device to host
        cudaMemcpy(P, d_P, matrix_size, cudaMemcpyDeviceToHost);

        //Free up the device memory matrices
        cudaFree(d_P);
        cudaFree(d_M);
        cudaFree(d_N);

}
```

MM (With Multiple Blocks)

- **Pd is broken into square tiles and all elements of a tile are computed by a block of threads**
- **Uses *blockIdx* to identify the tile and *threadIdx* to identify the element inside the tile**
- **All threads calculating the Pd elements within a tile have the same *blockIdx* values**
- **TILE_WIDTH is the dimensions of the block**
- **$Pd_{Row,\ Col}$ is inner product of the *Row* row of *Md* and *Col* column of *Nd*. Inner product is the sum of the products of the corresponding elements**

$$Pd_{Row,Col} = \sum d\_M_{Row,k} * d\_N_{k,Col},$$
$$for\ k = 0,1,\ldots Width - 1$$



Kernel Code

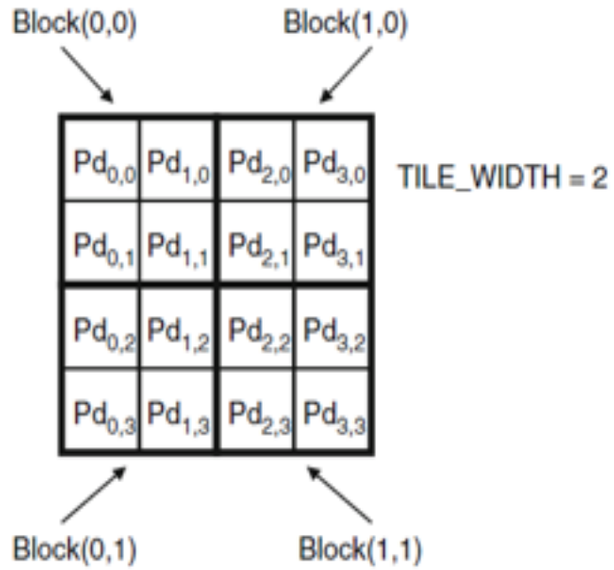
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
                                int Width)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float P_val = 0;

    for (int k = 0; k < Width; ++k) {
            float M_elem = d_M[row * Width + k];
            float N_elem = d_N[k * Width + col];
            P_val += M_elem * N_elem;
    }
    d_P[row*Width+col] = P_val;
}
```

Kernel Invocation(Slight Change)

```
int block_size = 16;

//Setup the  execution Configuration
dim3 dimGrid(Width/block_size, Width/block_size);
dim3 dimBlock(block_size, block_size);

//Launch the device computation threads
MatrixMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N, d_P, Width);
```

Example of using multiple blocks to calculate d_P

- Matrix divided into 4 blocks and each dimension is divided into sections of 2 elements
- Each block needs to calculate 4 elements
- Blocks organized into 2X2 arrays of threads
- Thread(0,0) of block(0,0) calculates $Pd_{0,0}$ where as thread(0,0) of block(1,0) calculates $Pd_{2,0}$
- bx = blockIdx.x, by = blockIdx.y, tx = threadId.x, ty = threadIdx.y,
  - For thread(0,0) and block(1,0)



Pd[bx * blockDim.x + tx][by * blockDim.y + ty] = Pd[1 * 2 + 0][0 * 2 + 0] = Pd[2][0]

Synchronization and Transparent Scalability

- ✦ CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function __synchthreads()
- ✦ When kernel function calls __synthreads(), all threads in a block will be held at the calling location until every thread in the block reaches the location.
- ✦ This ensures that all threads in a block have completed a phase of their execution
- ✦ These threads should execute in close time proximity with each other to avoid excessively long waiting times.

When a __syncthreads() statement is placed in an *if* statement,

Either all threads in a block execute the path that includes the __syncthreads() or none of them does.

For an *if–then–else* statement, if each path has a __syncthreads() statement,

Either all threads in a block execute the __syncthreads() on the *then* path or all of them execute the *else* path.

If a thread in a block executes the *then* path and another executes the *else* path

Would be waiting at different barrier synchronization points and may end up waiting for each other forever.

Tradeoff in design of CUDA Synchronization

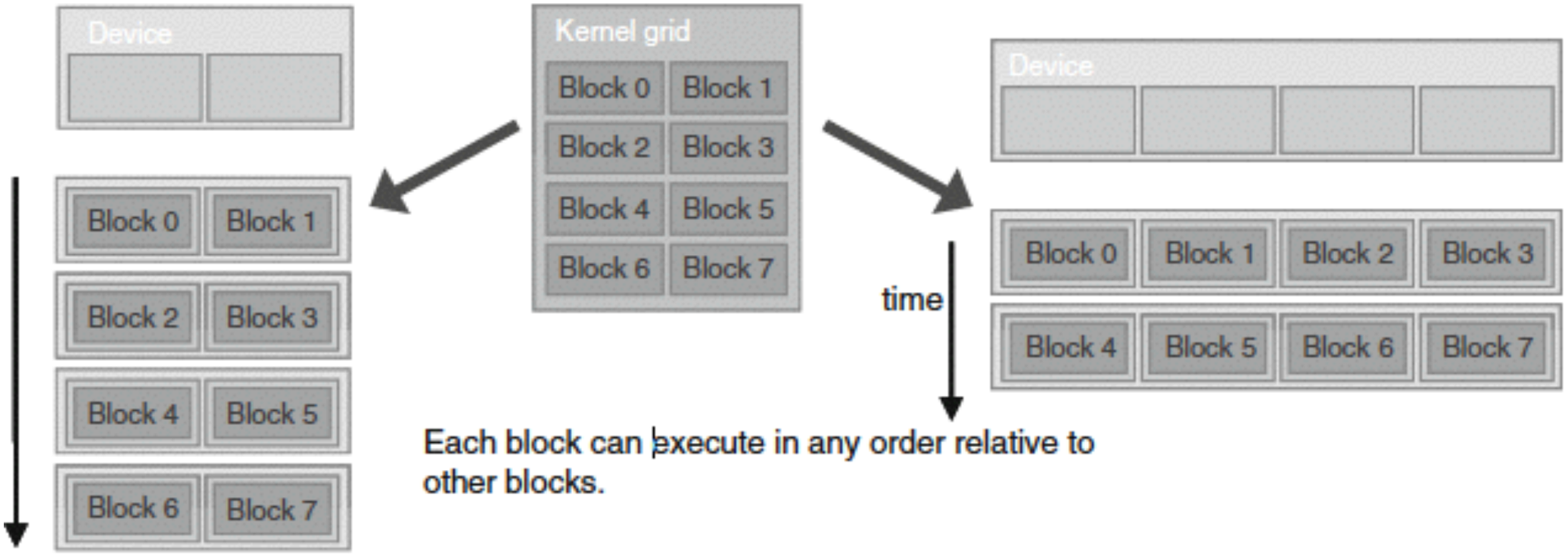


FIGURE 4.8

Transparent scalability for CUDA programs allowed by the lack of synchronization constraints between blocks.

- Ability to execute the same application code on hardware with different numbers of execution resources is referred to as transparent scalability.

Thread Assignment

- The execution resources are organized into streaming multiprocessors (SMs).
- Ex: Up to 8 blocks can be assigned to each SM in the GT200 design as long as there are enough resources to satisfy the needs of all of the blocks.
- In situations with an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit.
- The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.
- One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. Hardware resources are required for SMs to maintain the thread, block IDs, and track their execution status.

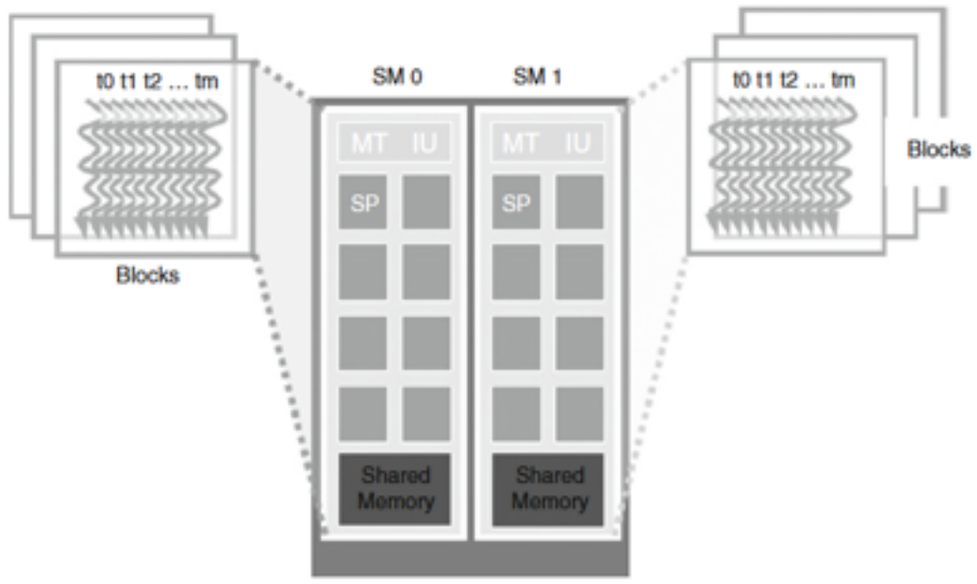


FIGURE 4.9
Thread block assignment to streaming multiprocessors (SMs).

- ✦ Once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called warps. The size of warps is implementation specific..
- ✦ We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM.
- ✦ If each block has 256 threads, then we can determine that each block has 256/32 or 8 warps. With 3 blocks in each SM, we have 8 x 3 =24 warps in each SM.
- ✦ When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results is selected for execution.
- ✦ This mechanism of filling the latency of expensive operations with work from other threads is often referred to as latency hiding.
- ✦ The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as zero-overhead thread scheduling.

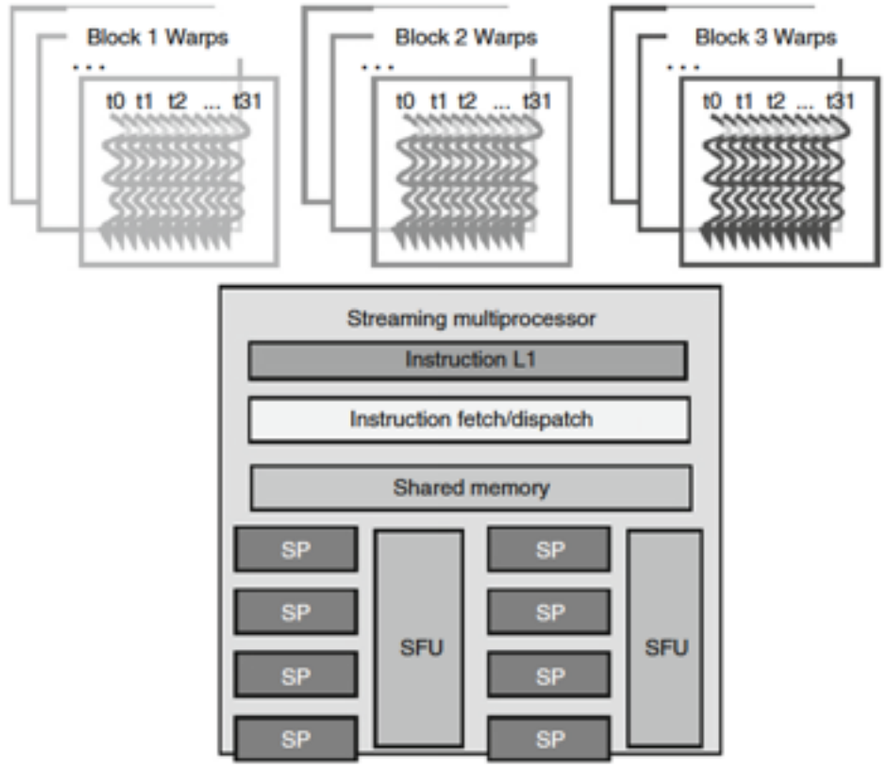


FIGURE 4.10
Blocks partitioned into warps for thread scheduling.