Chapter 3: Introduction to Data Parallelism and CUDA C

Data Parallelism

CUDA Program Structure

Vector Addition Kernel

Device Global Memory and Data Transfer

Kernel Functions and Threading

Data Parallelism

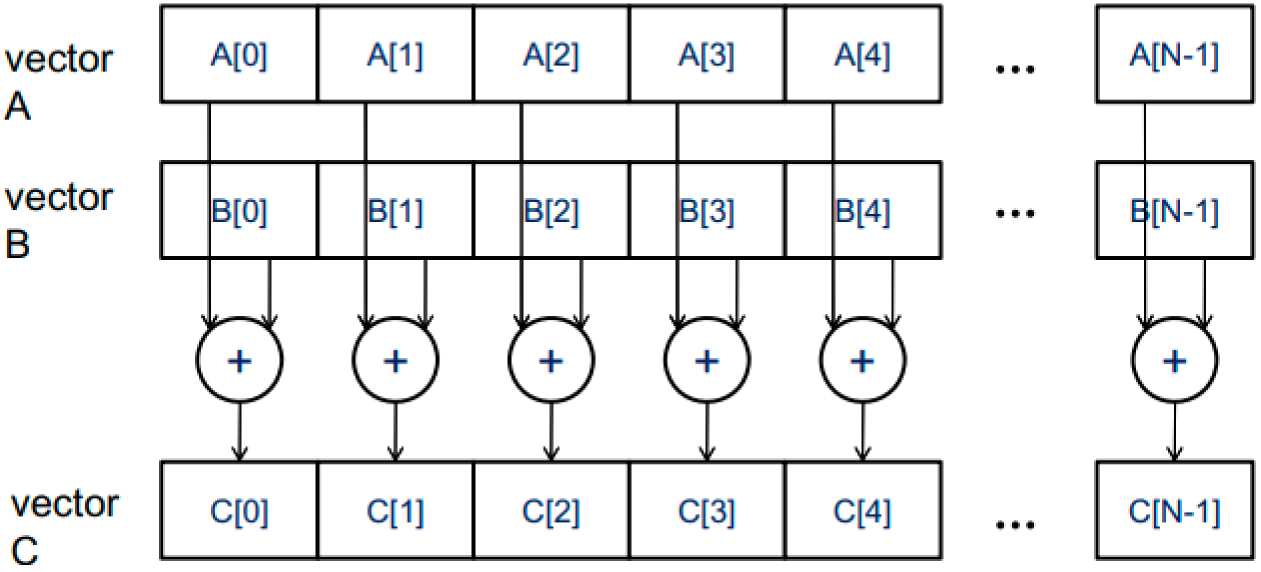


FIGURE 3.1

Data parallelism in vector addition.

CUDA PROGRAM STRUCTURE

- The structure of a CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer.
- Each CUDA source file can have a mixture of both host and device code.
- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device functions and data declarations into any C source file.
- The function or data declarations for the device are clearly marked with special CUDA keywords.

CUDA PROGRAM STRUCTURE

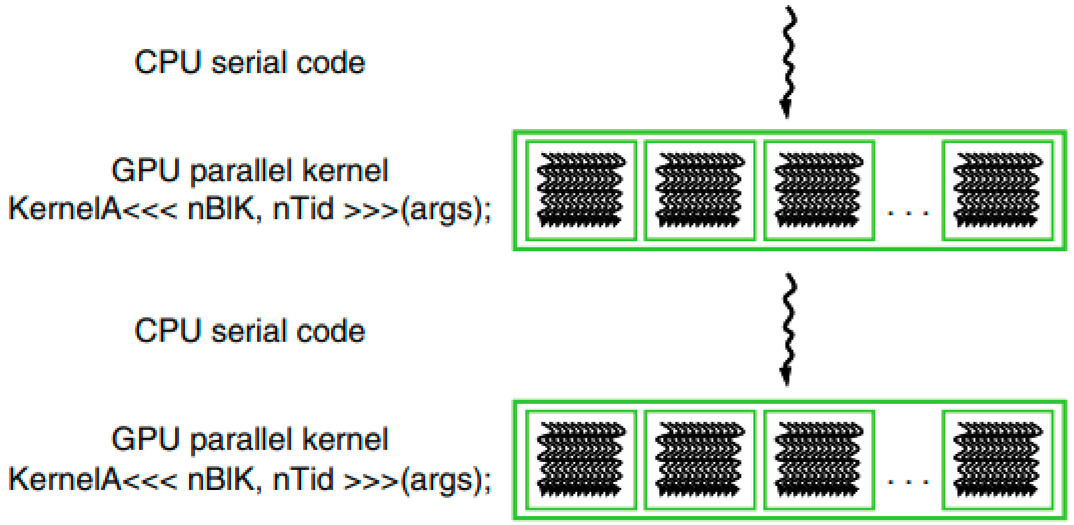
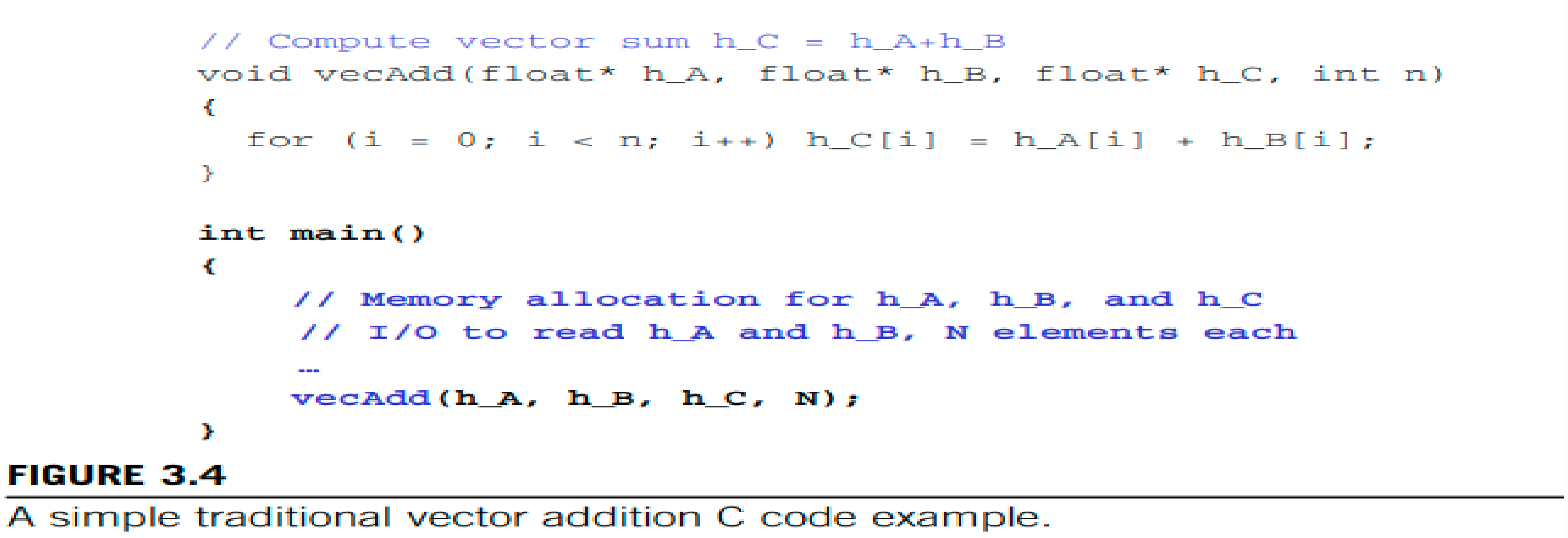


FIGURE 3.3

Execution of a CUDA program.

A VECTOR ADDITION KERNEL



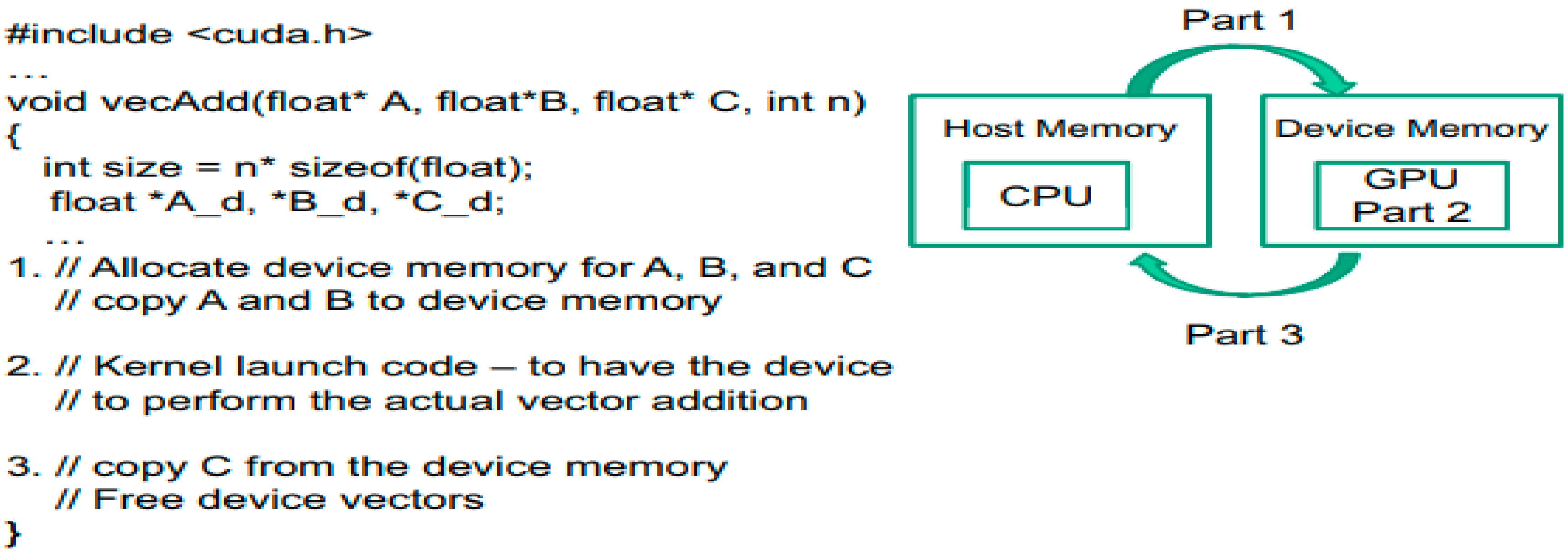
```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 3.4

A simple traditional vector addition C code example.

A straight forward way to execute vector addition in parallel is to modify the vecAdd() function and move its calculations to a CUDA device

Vector Addition Kernel



```
#include <cuda.h>
...
void vecAdd(float* A, float*B, float* C, int n)
{
   int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
   ...
1. // Allocate device memory for A, B, and C
    // copy A and B to device memory

2. // Kernel launch code – to have the device
    // to perform the actual vector addition

3. // copy C from the device memory
    // Free device vectors
}
```

Part 1
Host Memory
CPU
Device Memory
GPU
Part 2
Part 3

FIGURE 3.5

Outline of a revised `vecAdd()` function that moves the work to a device.

- At the beginning of the file, we need to add a C preprocessor directive to include the CUDA.h header file. This file defines the CUDA API functions and built-in variables
- Part 1 of the function allocates space in the device (GPU) memory to hold copies of the A, B, and C vectors, and copies the vectors from the host memory to the device memory. Part 2 launches parallel execution of the actual vector addition kernel on the device. Part 3 copies the sum vector C from the device memory back to the host memory.

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- In CUDA, host and devices have separate memory spaces. This reflects the current reality that devices are often hardware cards that come with their own DRAM. For example, the NVIDIA GTX480 comes with up to 4 GB (billion bytes, or gigabytes) of DRAM, called global memory

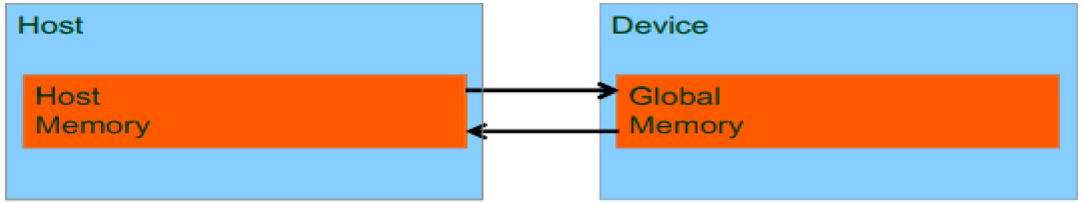
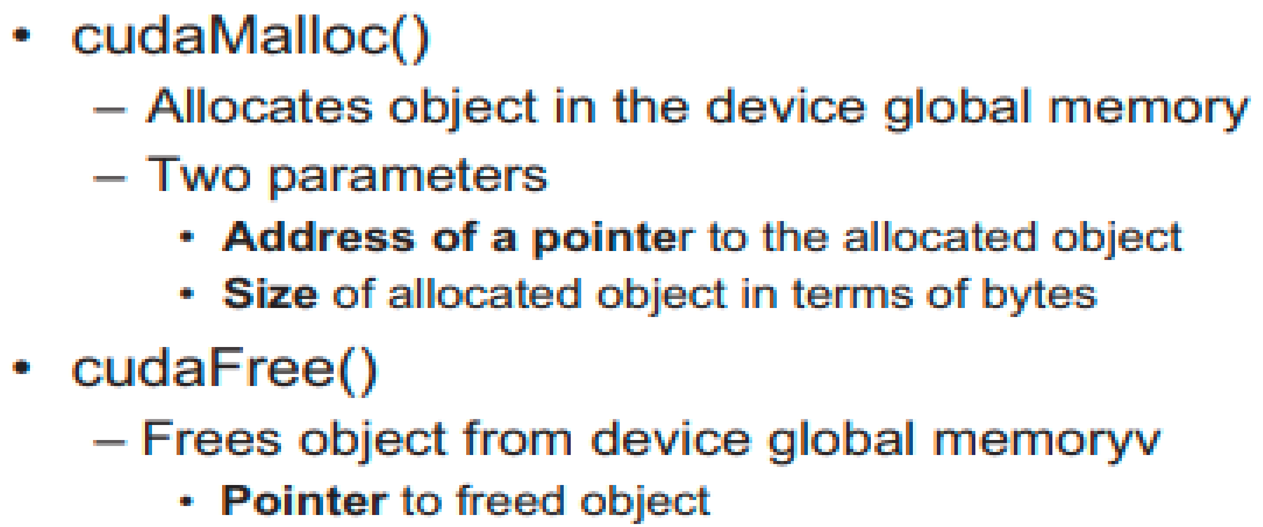


FIGURE 3.6

Host memory and device global memory.

- The CUDA runtime system provides Application Programming Interface (API) functions to perform these activities on behalf of the programmer. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory.

DEVICE GLOBAL MEMORY AND DATA TRANSFER



- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memoryv
    - **Pointer** to freed object

FIGURE 3.7

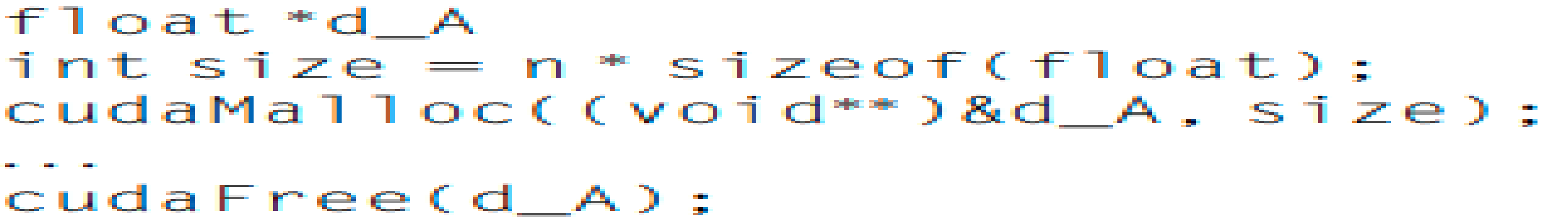
CUDA API functions for managing device global memory.

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Figure 3.7 shows two API functions for allocating and freeing device global memory. Function cudaMalloc() can be called from the host code to allocate a piece of device global memory for an object.
- The first parameter to the cudaMalloc() function is the address of a pointer variable that will be set to point to the allocated object.
- The address of the pointer variable should be cast to (void) because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.
- This parameter allows the cudaMalloc() function to write the address of the allocated memory into the pointer variable
- The second parameter to the cudaMalloc() function gives the size of the data to be allocated, in terms of bytes.

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- A simple code example to illustrate the use of cudaMalloc()

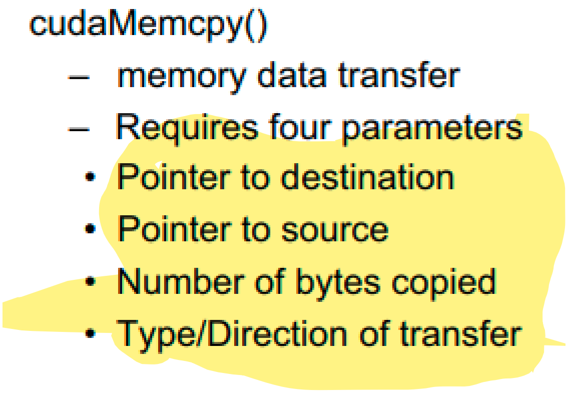


```
float *d_A
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

- The program passes the address of d_A (i.e., &d_A) as the first parameter after casting it to a void pointer. That is, d_A will point to the device memory region allocated for the A vector.
- The size of the allocated region will be n times the size of a single-precision floating number, which is 4 bytes in most computers today.
- After the computation, cudaFree() is called with pointer d_A as input to free the storage space for the A vector from the device global memory

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device.
- This is accomplished by calling one of the CUDA API functions. Figure 3.8 shows such an API function, cudaMemcpy()



cudaMemcpy()

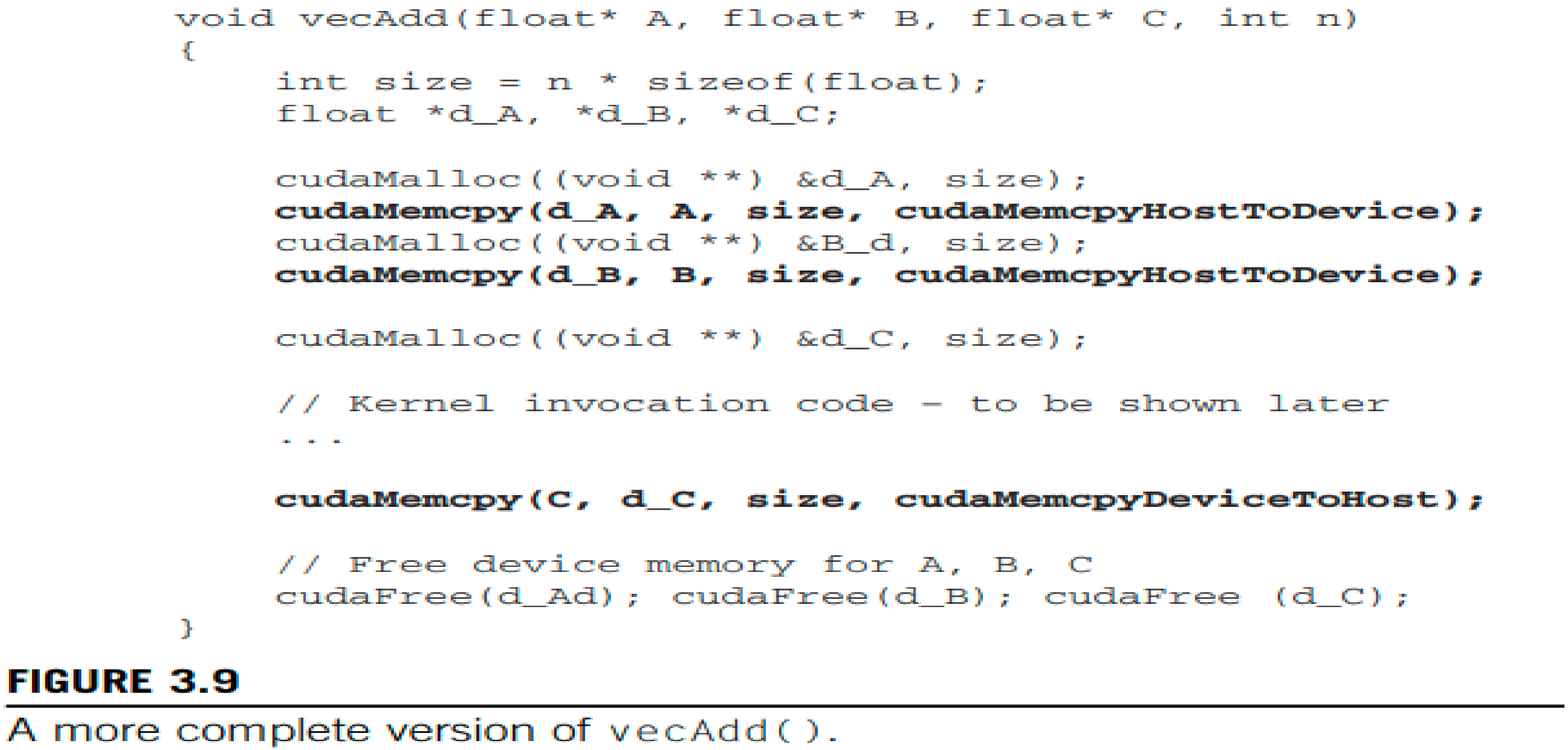
– memory data transfer

– Requires four parameters

- Pointer to destination
- Pointer to source
- Number of bytes copied
- Type/Direction of transfer

FIGURE 3.8

CUDA API function for data transfer between host and device.



```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later
    ...

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 3.9

A more complete version of vecAdd().

```
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

KERNEL FUNCTIONS AND THREADING

- In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Since all these threads execute the same code, CUDA programming is an instance of the well-known SPMD (single program, multiple data)
- The CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy.
- Grid is organized into an array of thread blocks
- All blocks of a grid are of the same size; each block can contain up to 1,024 threads

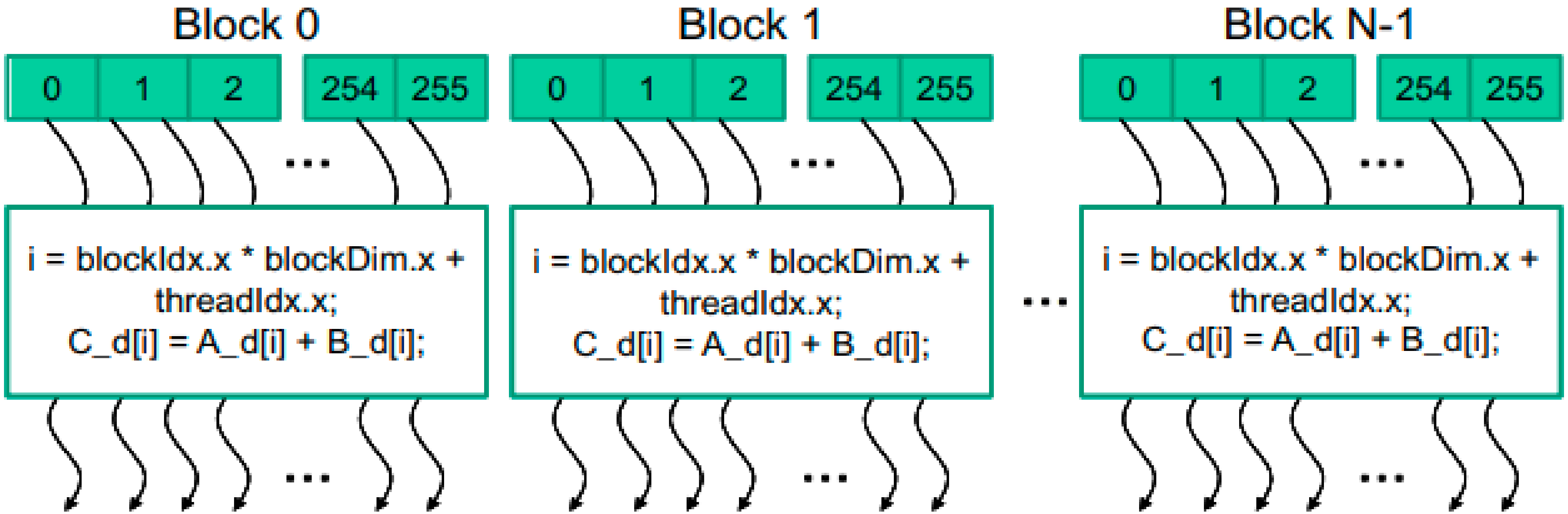
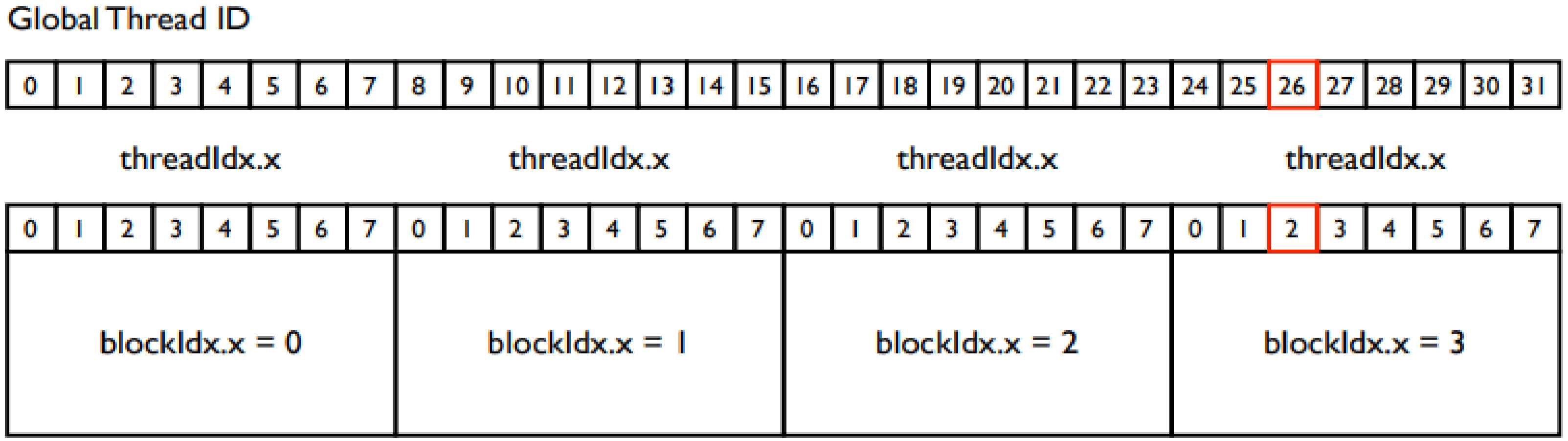

Block 0
0
1
2
254
255
Block 1
0
1
2
254
255
Block N-1
0
1
2
254
255
...
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

FIGURE 3.10

All threads in a grid execute the same kernel code.

KERNEL FUNCTIONS AND THREADING

- Each thread in a block has a unique **threadIdx** value
- This allows each thread to combine its **threadIdx** and **blockIdx** values to create a unique **global index** for itself with the entire grid



Global Thread ID

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

threadIdx.x threadIdx.x threadIdx.x threadIdx.x

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

blockIdx.x = 0 blockIdx.x = 1 blockIdx.x = 2 blockIdx.x = 3

KERNEL FUNCTIONS AND THREADING

- Figure 3.10 shows an example where each block consists of 256 threads.
- The number of threads in each thread block is specified by the host code when a kernel is launched.
- The same kernel can be launched with different numbers of threads at different parts of the host code.
- For a given grid of threads, the number of threads in a block is available in the blockDim variable.
- In Figure 3.10, the value of the blockDim.x variable is 256

blockDim speaks about the threads and similarly gridDim speaks about the blocks

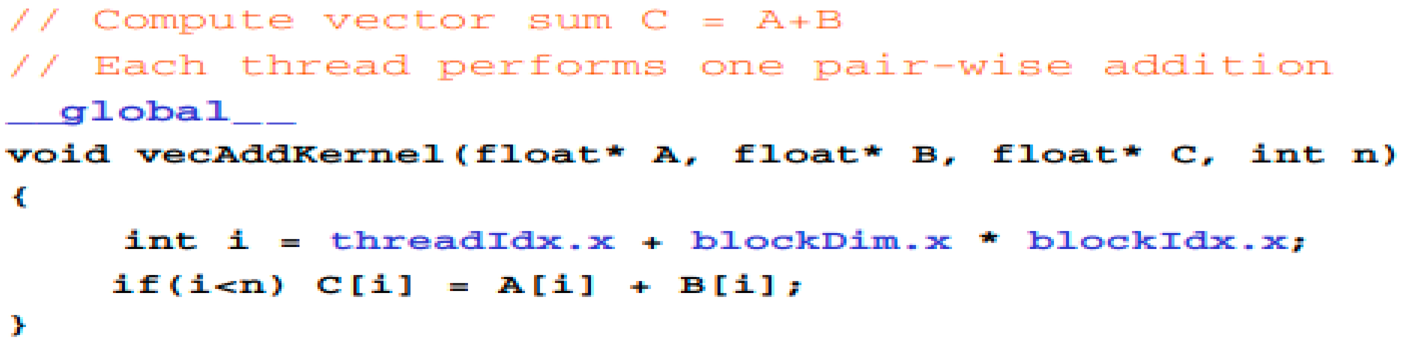
KERNEL FUNCTIONS AND THREADING

Id tells about the x and y values but the dim tells about how many in x and y for particular whole

- Each thread in a block has a unique *threadIdx* value.
- For example, the first thread in block 0 has value 0 in its *threadIdx* variable, the second thread has value 1, the third thread has value 2, etc.
- This allows each thread to combine its *threadIdx* and *blockIdx* values to create a unique global index for itself with the entire grid.
- In Figure 3.10, a data index i is calculated as *i= blockIdx.x * blockDim.x + threadIdx.x*.
- Since blockDim is 256 in our example, the i values of threads in block 0 ranges from 0 to 255. The i values of threads in block 1 range from 256 to 511. The i values of threads in block 2 range from 512 to 767.
- Since each thread uses i to access d_A, d_B, and d_C, these threads cover the first 768 iterations of the original loop. By launching the kernel with a larger number of blocks, one can process larger vectors.

KERNEL FUNCTIONS AND THREADING

- __global__ indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device



```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

FIGURE 3.11

A vector addition kernel function and its launch statement.

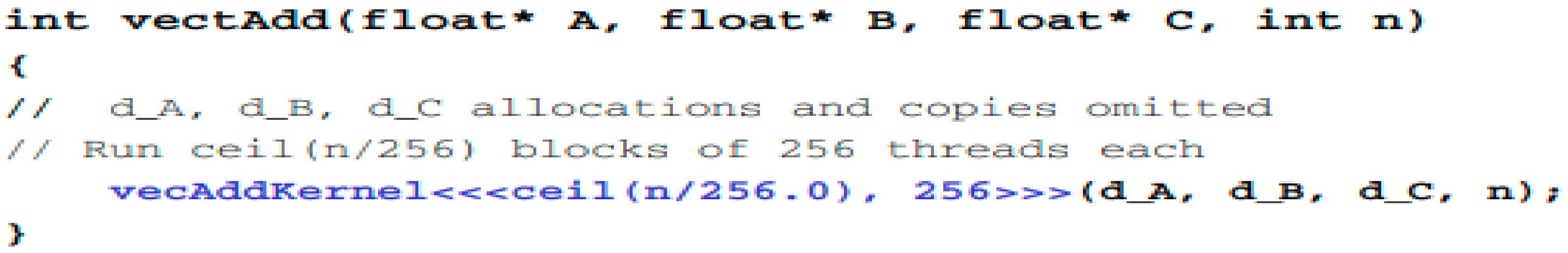
- In a CUDA kernel function, *automatic (local) variables i are private to each thread*. That is, a version of *i* will be generated for every thread. If the kernel is launched with 10,000 threads, there will be 10,000 versions of *i*, one for each thread.

all the variables inside the kernel function are local to each thread i.e kernel function is run by each thread so each thread makes local value of the variable and use it

KERNEL FUNCTIONS AND THREADING

importance of passing size and use of i<n(size) and all

- *Not* all vector lengths can be expressed as multiples of the block size.
- For example, if the vector length is 100, the smallest efficient thread block dimension is 32.
- Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements.
- However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program.
- Since all threads are to execute the same code, all will test their *i* values against *n*, which is *100*. With the *if (i<n)* statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.



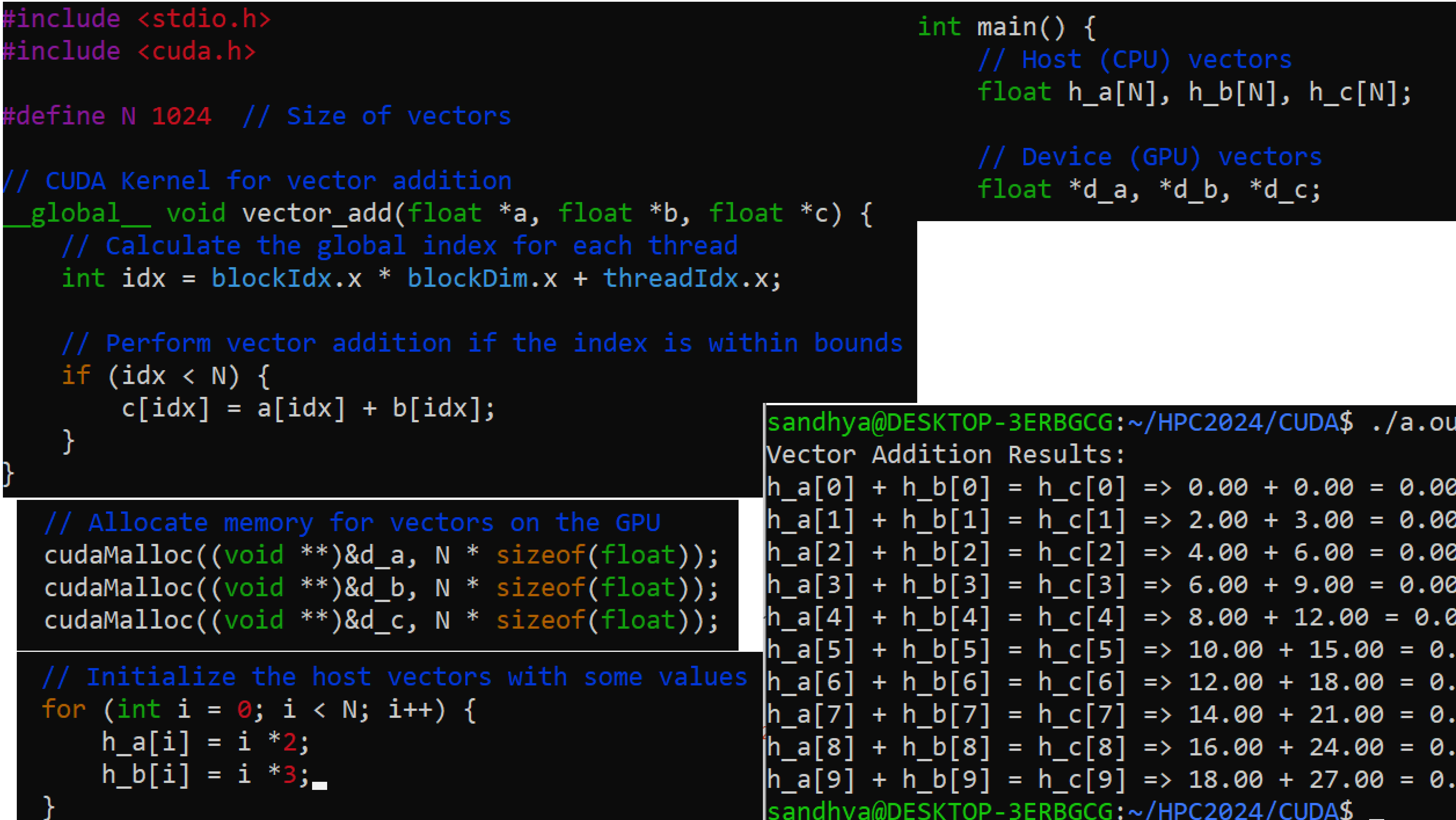
```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

FIGURE 3.13

A vector addition kernel function and its launch statement.

- When the host code launches a kernel, it sets the grid and thread block dimensions via execution configuration parameters.
- This is illustrated in Figure 3.13.
- The configuration parameters are given between the <<<and >>> before the traditional C function arguments.
- The first configuration parameter gives the **number of thread blocks in the grid**. The second specifies **the number of threads in each thread block**.
- In this example, there are 256 threads in each block. To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0.
- Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1,000 threads, we would launch ceil(1,000/256.0)=4 thread blocks. As a result, the statement will launch 4 *256=1,024 threads.

to use ceil we need to use the floating point so we do use 256.0 and then do the work



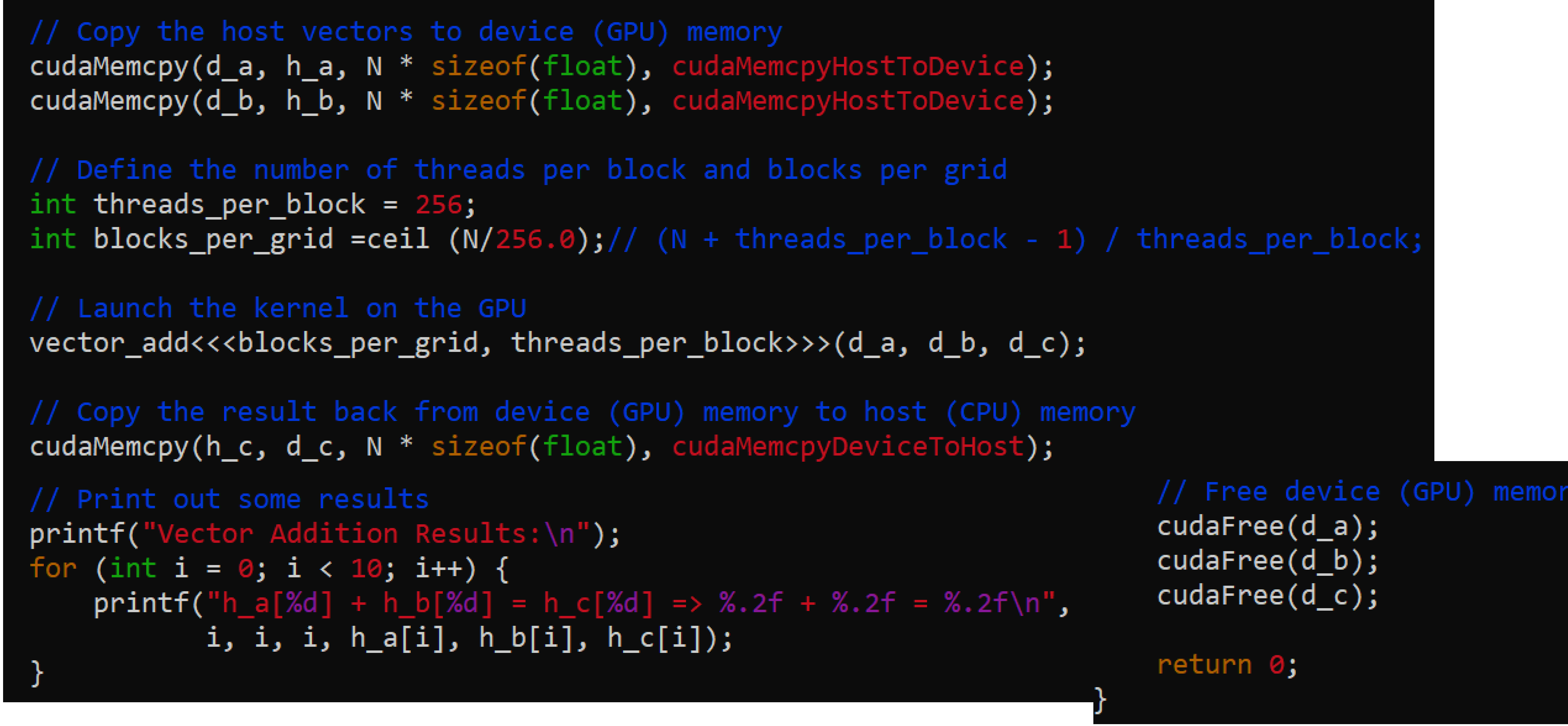
```
#include <stdio.h>
#include <cuda.h>

#define N 1024  // Size of vectors

// CUDA Kernel for vector addition
__global__ void vector_add(float *a, float *b, float *c) {
    // Calculate the global index for each thread
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Perform vector addition if the index is within bounds
    if (idx < N) {
        c[idx] = a[idx] + b[idx];
    }
}
```

```
int main() {
    // Host (CPU) vectors
    float h_a[N], h_b[N], h_c[N];

    // Device (GPU) vectors
    float *d_a, *d_b, *d_c;
```

```
  // Allocate memory for vectors on the GPU
  cudaMalloc((void **)&d_a, N * sizeof(float));
  cudaMalloc((void **)&d_b, N * sizeof(float));
  cudaMalloc((void **)&d_c, N * sizeof(float));
```

```
  // Initialize the host vectors with some values
  for (int i = 0; i < N; i++) {
      h_a[i] = i *2;
      h_b[i] = i *3;
  }
```

```
sandhya@DESKTOP-3ERBGCG:~/HPC2024/CUDA$ ./a.ou
Vector Addition Results:
h_a[0] + h_b[0] = h_c[0] => 0.00 + 0.00 = 0.00
h_a[1] + h_b[1] = h_c[1] => 2.00 + 3.00 = 0.00
h_a[2] + h_b[2] = h_c[2] => 4.00 + 6.00 = 0.00
h_a[3] + h_b[3] = h_c[3] => 6.00 + 9.00 = 0.00
h_a[4] + h_b[4] = h_c[4] => 8.00 + 12.00 = 0.0
h_a[5] + h_b[5] = h_c[5] => 10.00 + 15.00 = 0.
h_a[6] + h_b[6] = h_c[6] => 12.00 + 18.00 = 0.
h_a[7] + h_b[7] = h_c[7] => 14.00 + 21.00 = 0.
h_a[8] + h_b[8] = h_c[8] => 16.00 + 24.00 = 0.
h_a[9] + h_b[9] = h_c[9] => 18.00 + 27.00 = 0.
sandhya@DESKTOP-3ERBGCG:~/HPC2024/CUDA$
```



```
// Copy the host vectors to device (GPU) memory
cudaMemcpy(d_a, h_a, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, N * sizeof(float), cudaMemcpyHostToDevice);

// Define the number of threads per block and blocks per grid
int threads_per_block = 256;
int blocks_per_grid =ceil (N/256.0);// (N + threads_per_block - 1) / threads_per_block;

// Launch the kernel on the GPU
vector_add<<<blocks_per_grid, threads_per_block>>>(d_a, d_b, d_c);

// Copy the result back from device (GPU) memory to host (CPU) memory
cudaMemcpy(h_c, d_c, N * sizeof(float), cudaMemcpyDeviceToHost);

// Print out some results
printf("Vector Addition Results:\n");
for (int i = 0; i < 10; i++) {
    printf("h_a[%d] + h_b[%d] = h_c[%d] => %.2f + %.2f = %.2f\n",
           i, i, i, h_a[i], h_b[i], h_c[i]);
}
```

```
    // Free device (GPU) memor
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Kernel Functions and Threading

__global__

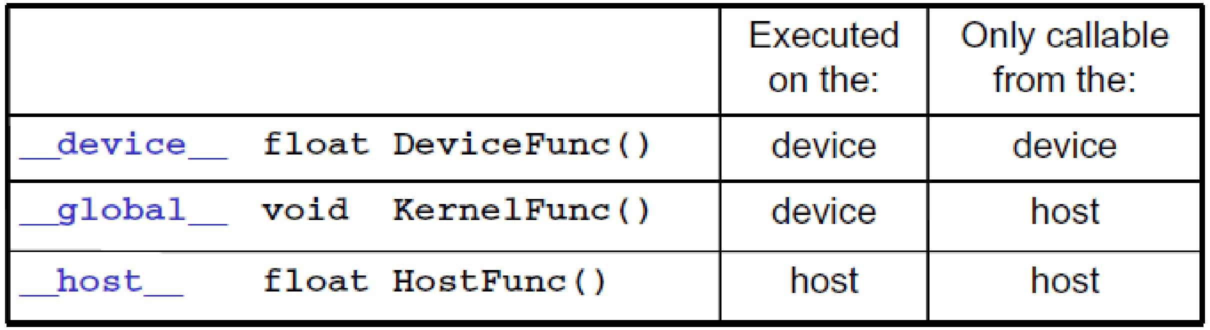
- Keyword indicates that the function being declared is a CUDA kernel function.
- Function is to be executed on the device and can only be called from the host code.

__device__

- Keyword indicates that the function being declared is a CUDA device function.
- A device function executes on a CUDA device and can only be called from a kernel function or another device function.

__host__

- Host function is simply a traditional C function that executes on the host and can only be called from another host function



	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 3.12

CUDA C keywords for function declaration.

Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION

- Threads are organized into a two-level hierarchy:
  - a grid consists of one or more blocks and
  - each block in turn consists of one or more threads.
- All threads in a block share the same block index, which can be accessed as the blockIdx variable in a kernel.
- Each thread also has a thread index, which can be accessed as the threadIdx variable in a kernel.
- To a CUDA programmer, blockIdx and threadIdx appear as built-in variables that can be accessed within kernel functions

Chapter 4:Data Parallel Execution Model

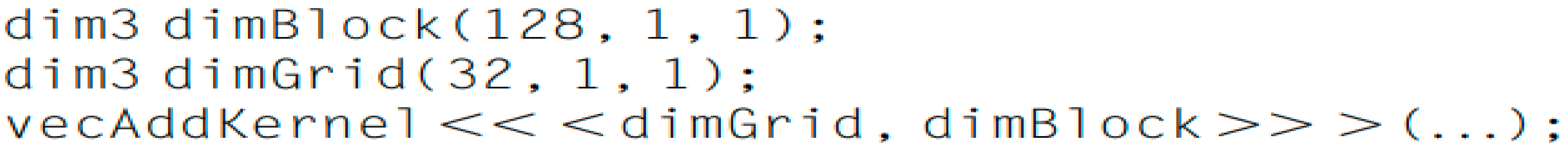
CUDA THREAD ORGANIZATION

what we generall specify in execution config <<< >>> is the gridDim and blockDim actually

- The execution configuration parameters in a kernel launch statement specify the dimensions of the grid and the dimensions of each block
- These dimensions are available as predefined built-in variables gridDim and blockDim in kernel functions
- A grid is a 3D array of blocks and each block is a 3D array of threads.
- The programmer can choose to use fewer dimensions by setting the unused dimensions to 1.
- The exact organization of a grid is determined by the execution configuration parameters (within <<< and >>>) of the kernel launch statement.
- Each such parameter is of dim3 type, which is a C struct with three unsigned integer fields, x, y, and z. These three fields correspond to the three dimensions. For 1D or 2D grids and blocks, the unused dimension fields should be set to 1 for clarity

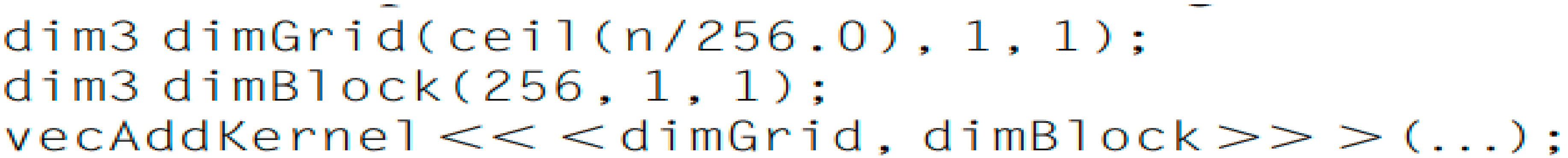
Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION



```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(32, 1, 1);
vecAddKernel<< <dimGrid, dimBlock>> >(...);
```

- The grid and block dimensions can also be calculated from other variables. For example, the kernel launch in Figure 3.14 can be written as:



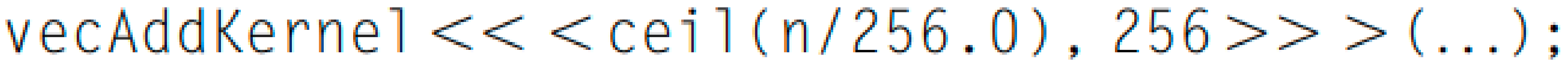
```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<< <dimGrid, dimBlock>> >(...);
```

- This allows *the number of blocks to vary with the size of the vectors* so that the grid will have enough threads to cover all vector elements.
- The value of variable *n* at kernel launch time will determine the dimension of the grid.
- If n is equal to 1,000, the grid will consist of four blocks. If n is equal to 4,000, the grid will have 16 blocks

Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION

- For convenience, CUDA C provides a special shortcut for launching a kernel with 1D grids and blocks.
- Instead of using dim3 variables, one can use arithmetic expressions to specify the configuration of 1D grids and blocks.



```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

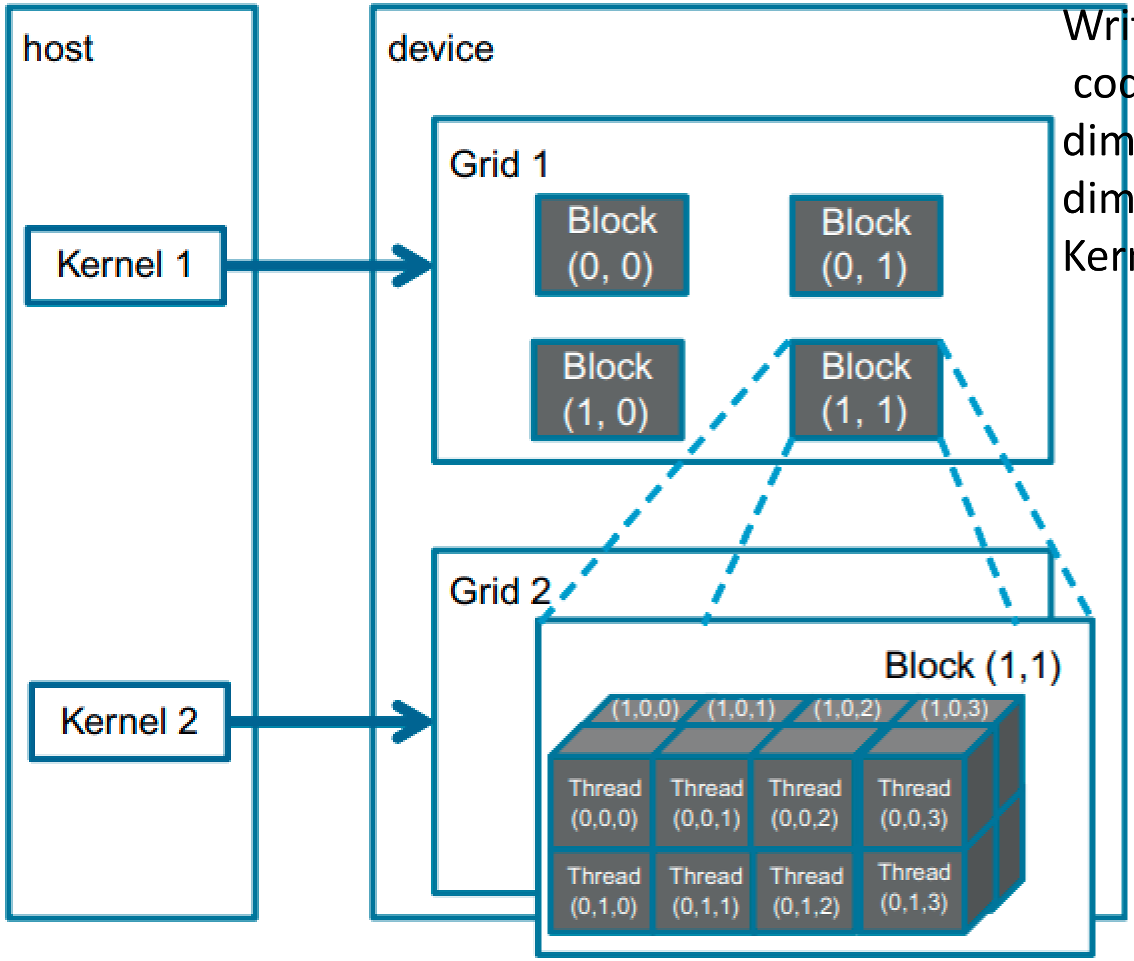
- In this case, the CUDA C compiler simply takes the arithmetic expression as the x dimensions and assumes that the y and z dimensions are 1. This gives us the kernel launch statement shown in Figure 3.14
- In CUDA C, the allowed values of gridDim.x, gridDim.y, and gridDim.z range from 1 to 65,536.
- All threads in a block share the same blockIdx.x, blockIdx.y, and blockIdx.z values. Among all blocks, the blockIdx.x value ranges between 0 and gridDim.x-1, the blockIdx.y value between 0 and gridDim.y-1, and the blockIdx.z value between 0 and gridDim.z-1

Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION: BLOCK

- Blocks are organized into 3D arrays of threads.
- Two-dimensional blocks can be created by setting the z dimension to 1. One-dimensional blocks can be created by setting both the y and z dimensions to 1
- The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024. max no of threads in the block is 1024 i.e blockDIm can have max of total of 1024 spread across x,y and z
- For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable blockDim values,
- But (32, 32, 2) is not allowable since the total number of threads would exceed 1,024
- Note that the grid can have higher dimensionality than its blocks and vice versa

Chapter 4:Data Parallel Execution Model



Write the host code to generate the Grid 1 of Fig.4.1

code:

```
dim3 dimBlock(4, 2, 2);
dim3 dimGrid(2, 2, 1);
KernelFunction <<< dimGrid, dimBlock>>>(...);
```

example where the dimensionality of the blocks is greater then the dimensionality of the grid

FIGURE 4.1

A multidimensional example of CUDA grid organization.

Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION:

- The grid consists of four blocks organized into a 2 * 2 array.
- Each block in Figure 4.1 is labeled with *(blockIdx.y, blockIdx.x).*
- For example, block(1,0) has blockIdx.y = 1 and blockIdx.x = 0.
- Note that the ordering of the labels is such that the highest dimension comes first.
- This is reverse of the ordering used in the configuration parameters where the lowest dimension comes first. This reversed ordering for labeling threads works better when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional arrays.

so when dim3 it is x,y,z but when naming each block or thread and all it is higher dim comes first so whenever given labelling take higher as first and proceed like if(4,3) take 4 as y and 3 as x and then proceed

- Each threadIdx also consists of three fields: the x coordinate threadId.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z.
- Figure 4.1 illustrates the organization of threads within a block. In this example, each block is organized into 4 * 2 * 2 arrays of threads.

block id tells location of block in the grid and threadid will tell the location of thread in the block with the help of x,y,z values

Chapter 4:Data Parallel Execution Model

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data.
- For example, pictures are a 2D array of pixels.
- It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture.
- Figure 4.2 shows such an arrangement for processing a 76 * 62 picture (76 pixels in the horizontal or x direction and 62 pixels in the vertical or y direction).
- Assume that we decided to use a 16 * 16 block, with 16 threads in the x direction and 16 threads in the y direction.
- We will need five blocks in the x direction and four blocks in the y direction, which results in 5 * 4 = 20 blocks as shown in Figure 4.2.

Chapter 4:Data Parallel Execution Model

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- The heavy lines mark block boundaries. shaded area depicts threads that cover pix Note that we have four e threads in the x direction two extra threads in th direction. That is, we generate 80 * 64 thread process 76 * 62 pixels

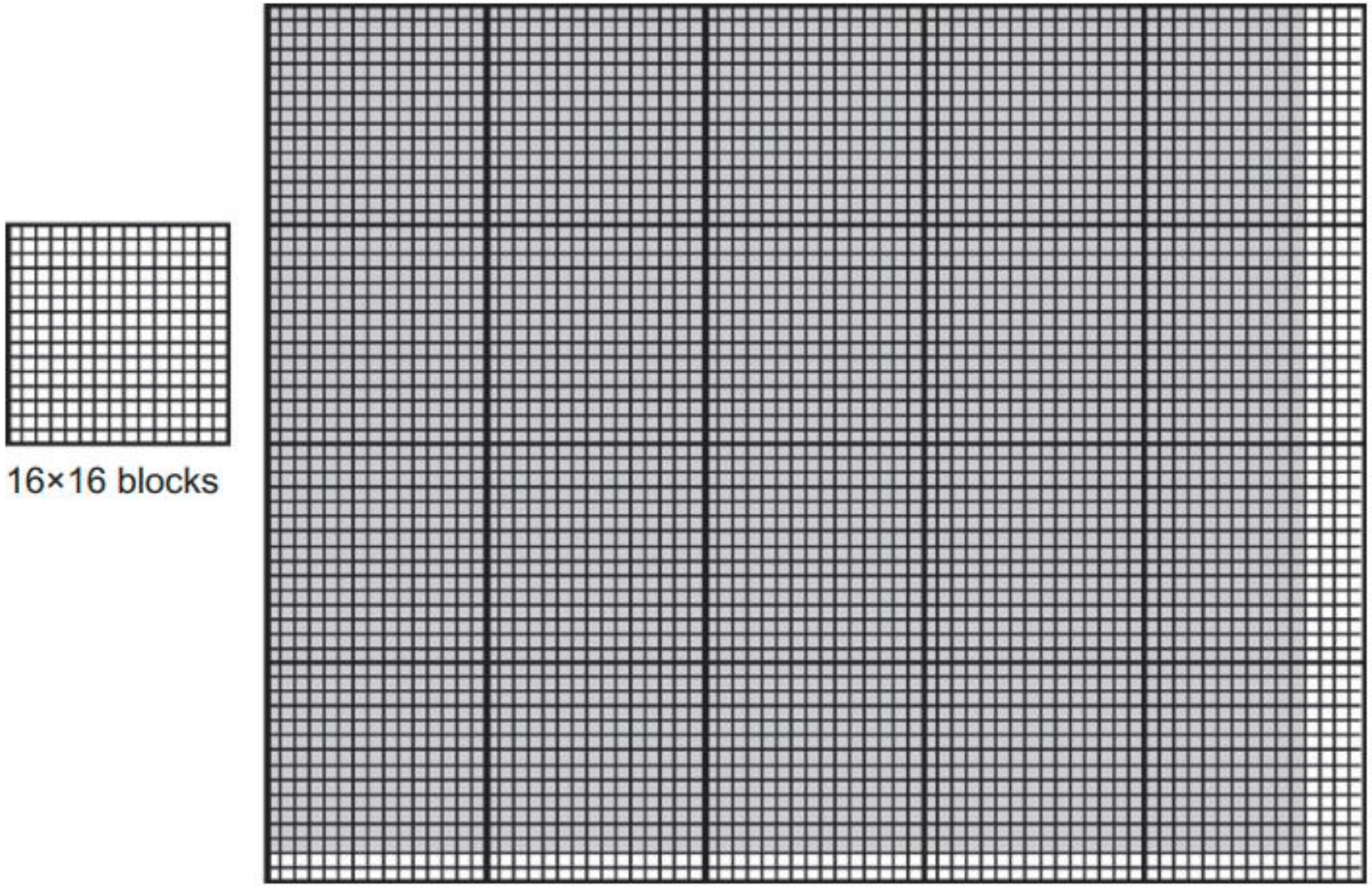


FIGURE 4.2

Using a 2D grid to process a picture.

Chapter 4:Data Parallel Execution Model

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Following host code can be used to launch a 2D kernel to process the picture:

```
dim3 dimBlock(16,16,1);
dim3 dimGrid(ceil(n/16.0), ceil(m/16.0), 1);
pictureKernel <<<dimGrid, dimBlock >>>(d_Pin, d_Pout, n, m);
```

- The dimensions of the grid depend on the dimensions of the picture.
- To process a 2,000 * 1,500 (3 M pixel) picture, How many block is required?
- 11750 blocks, 125 in the x direction and 94 in the y direction.
- Within the kernel function, references to built-in variables gridDim.x, gridDim.y, blockDim.x, and blockDim.y will result in ………………………………..
- 125, 94, 16, and 16, respectively

Chapter 4:Data Parallel Execution Model

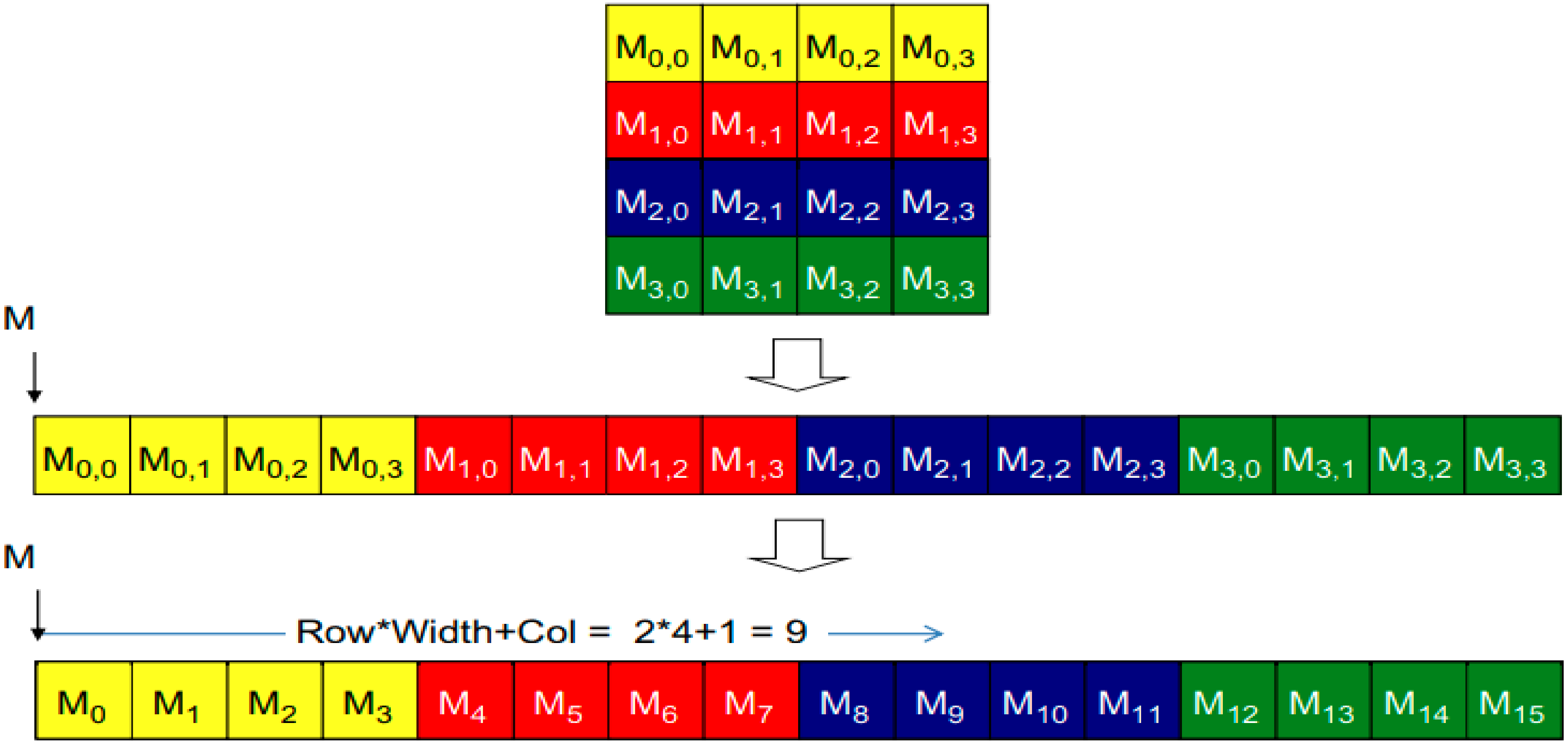
MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Most modern computers have at least 4 GB-sized locations,
- where each G is 1,073,741,824 (230).
- All locations are labeled with an address that ranges from 0 to the largest number.
- Since there is only one address for every location, we say that the memory space has a “flat” organization.
- So, all multidimensional arrays are ultimately “flattened” into equivalent 1D arrays. While a C programmer can use a multidimensional syntax to access an element of a multidimensional array, the compiler translates these accesses into a base pointer that points to the beginning element of the array, along with an offset calculated from these multidimensional indices.
- All multidimensional arrays in C are linearized. This is due to the use of a “flat” memory space in modern computers

Chapter 4:Data Parallel Execution Model

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- In the case of dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers due to lack of dimensional information.
- There are at least two ways one can linearize a 2D array.
  - One is to place all elements of the same row into consecutive locations. The rows are then placed one after another into the memory space. This arrangement, called row-major layout, is illustrated in Figure 4.3.



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

FIGURE 4.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression `Row*Width + Col` for an element that is in the `Row`th row and `Col`th column of an array of `Width` elements in each row.

in row major it is row(i)*no of columns+j(col)

Chapter 4:Data Parallel Execution Model

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- To increase the readability, we will use Mj,i to denote an M element at the j row and the i column. Mj,i is equivalent to the C expression M[j][i] but slightly more readable. Figure 4.3 shows an example where a 4 * 4 matrix M is linearized into a 16-element 1D array, with all elements of row 0 first, followed by the four elements of row 1, etc.
- For example, the 1D index for M2,1 is 2 * 4 + 1 = 9. This is illustrated in Figure 4.3, where M9 is the 1D equivalent to M2,1. This is the way C compilers linearize 2D arrays.
- Another way to linearize a 2D array is to place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space. This arrangement, called column major layout, is used by FORTRAN compilers. Note that the column major layout of a 2D array is equivalent to the row-major layout of its transposed form.
- CUDA C uses row-major layout rather than column-major layout

CUDA Thread Organization

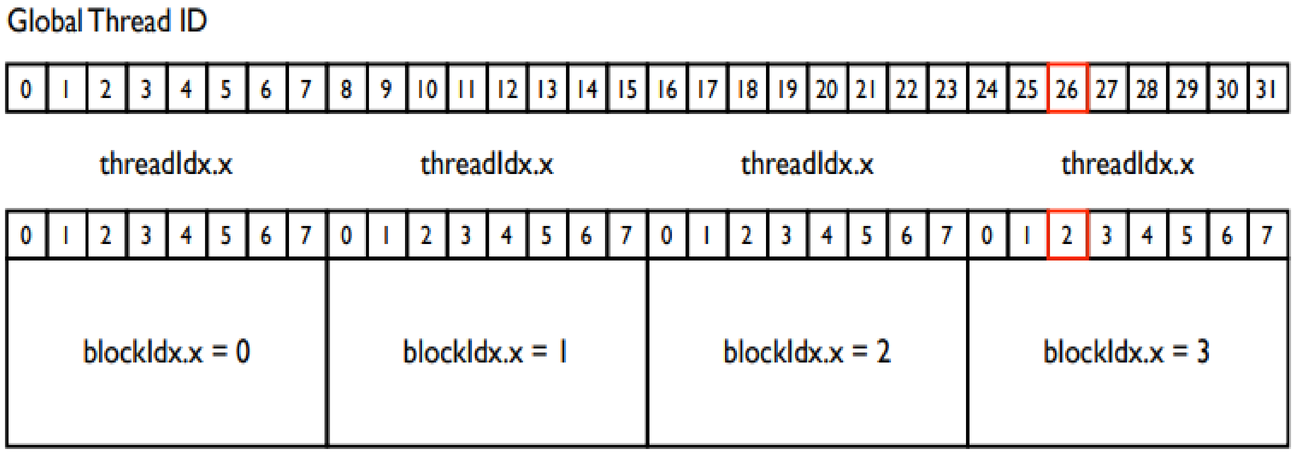
- A thread block can be organized in 1D, 2D, or 3D.

1D Blocks and 1D Grid:

- localThreadID = threadIdx.x
- globalThreadID = blockIdx.x * blockDim.x + threadIdx.x (Global thread ID is calculated by considering how many threads are in the previous blocks and then adding the thread index in the current block)

Example:

- blockDim.x = 4 (4 threads per block)
- gridDim.x = 3 (3 blocks)
- For blockIdx.x = 2 and threadIdx.x = 3
- globalThreadID = 2 * 4 + 3 = 11



CUDA Thread Organization

2D Blocks and 2D Grid:

- localThreadID = threadIdx.y * blockDim.x + threadIdx.x
- globalThreadID = (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.x * blockDim.y) + threadIdx.y * blockDim.x + threadIdx.x (Global thread ID is calculated by considering total number of threads from previous blocks (both x and y) and adding the current thread's position)

Example:

- blockDim.x = 4, blockDim.y = 2 (8 threads per block)
- gridDim.x = 3, gridDim.y = 2 (6 blocks)
- For blockIdx.x = 2, blockIdx.y = 1 and threadIdx.x = 3, threadIdx.y = 1:
- globalThreadID = (1 * 3 + 2) * 8 + 1 * 4 + 3 = 40 + 7 = 47

CUDA Thread Organization

3D Blocks and 3D Grid:

- localThreadID = threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x
- globalThreadID = (blockIdx.z * gridDim.y * gridDim.x + blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.z * blockDim.y * blockDim.x)) + (threadIdx.z * blockDim.y * blockDim.x) + (threadIdx.y * blockDim.x) + threadIdx.x
- (Global thread ID is calculated by considering block's position in all three dimensions and the thread's position within the block:)

Example:

- blockDim.x = 4, blockDim.y = 2, blockDim.z = 2 (16 threads per block)
- gridDim.x = 3, gridDim.y = 2, gridDim.z = 2 (12 blocks)
- For blockIdx.x = 2, blockIdx.y = 1, blockIdx.z = 1 and threadIdx.x = 3, threadIdx.y = 1, threadIdx.z = 1:
- globalThreadID = ((1 * 2 * 3 + 1 * 3 + 2) * 16) + (1 * 8) + (1 * 4) + 3

  = 112 + 8 + 4 + 3

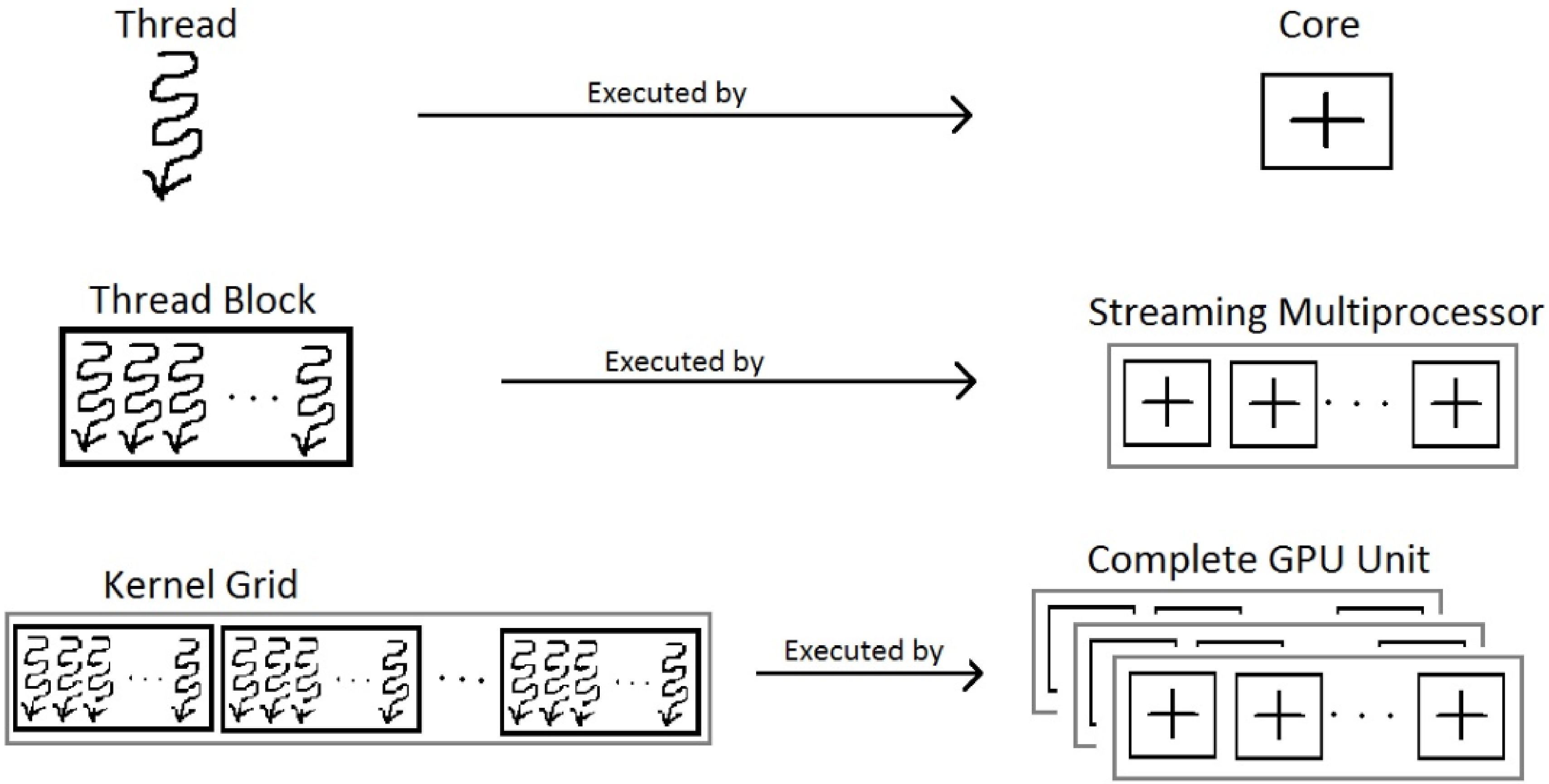
  = 127

Streaming Multiprocessors

threads -> wraps -> blocks -> sm -> gpu

- Hardware groups several threads that execute the same instructions into **wraps**
- Several wraps constitute a thread block
- Several thread blocks are assigned to a Streaming Multiprocessors (SM)
- Several SM constitute a GPU
- SM: These are general purpose processors with low clock rate and small cache
- The primary task of an SM is that it must execute several thread blocks in parallel
- As soon as one of its thread block has completed execution, it takes up the serially next thread block

Streaming Multiprocessors



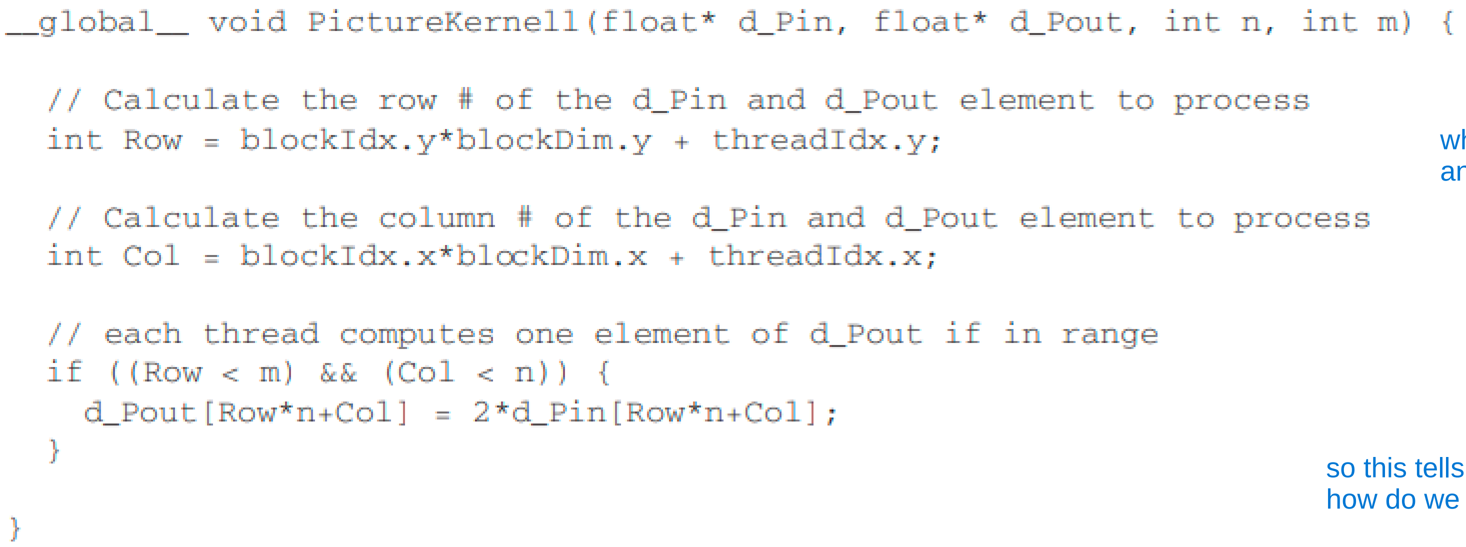
Streaming Multiprocessors

1. **Execution cores:** single precision floating-point units, double precision floating-point units, special function units (SFUs)
2. **Caches**
   1. L1 cache. (for reducing memory access latency).
   2. Shared memory (for shared data between threads).
   3. Constant cache (for broadcasting of reads from a read-only memory).
   4. Texture cache (for aggregating bandwidth from texture memory).
3. **Schedulers for warps** :these are for issuing instructions to warps based on a particular scheduling policies
4. **A substantial number of registers**: an SM may be running a large number of active threads at a time, so it is a must to have registers in thousands

Chapter 4:Data Parallel Execution Model

CUDA THREAD ORGANIZATION: BLOCK

source code of pictureKernel(), shown in Figure 4.4. Let’s assume that the kernel will scale every pixel value in the picture by a factor of 2.0. The kernel code is conceptually quite simple. There are a total of blockDim.x*gridDim.x threads in the horizontal direction.



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

  // 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 process
  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];
  }

}
```

when in 2d how do you access each row and col the i and j is like this

so this tells how to access when we do each 2d to grid each ele how do we access it as a matrix and then do

FIGURE 4.4

Source code of pictureKernel() showing a 2D thread mapping to a data pattern.

here no concept of that formula of local and global thread id use comes that is when we need a for whole a unique id we do that

Chapter 4:Data Parallel Execution Model

This Fig. illustrates the execution of pictureKernel() when processing our 76 * 62 example.

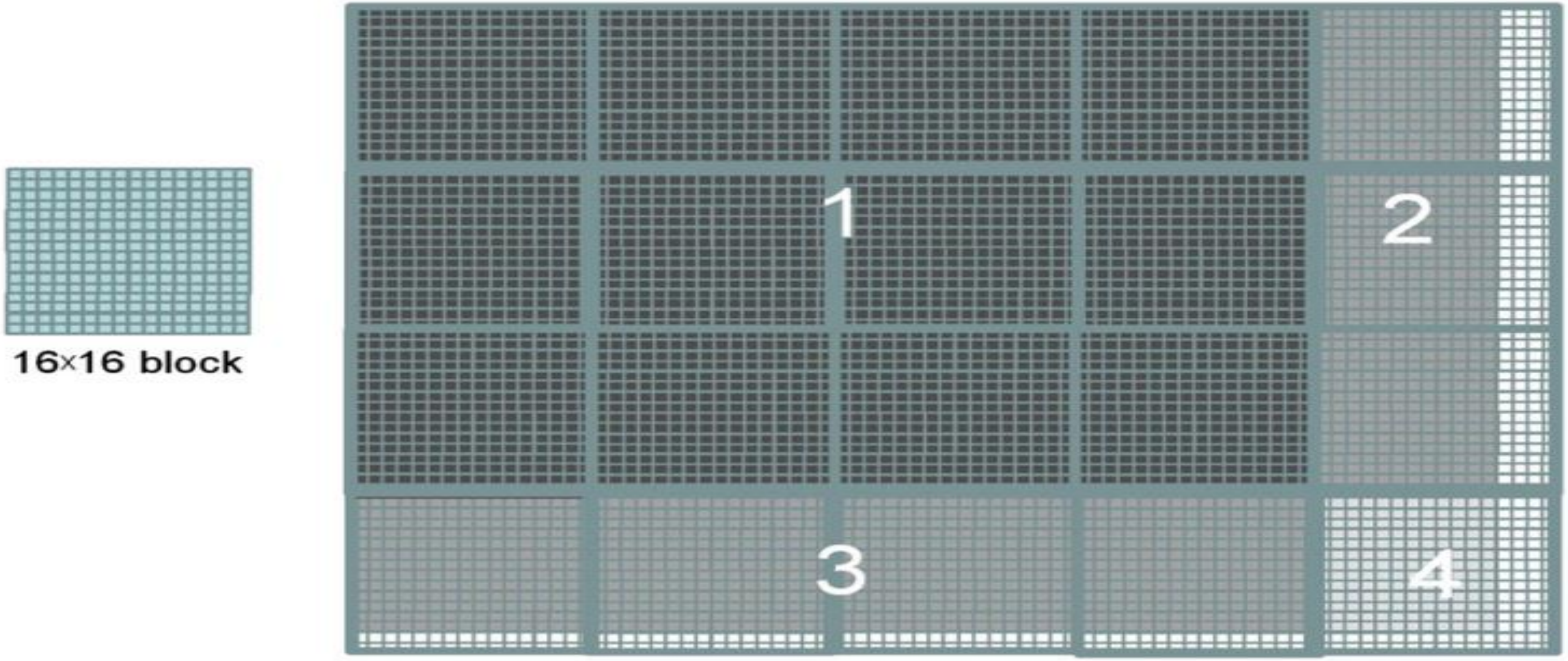


FIGURE 4.5

Covering a 76 × 62 picture with 16 × 16 blocks.

Chapter 4:Data Parallel Execution Model

We have studied vecAddkernel() and pictureKernel() where each thread performs only one floating-point arithmetic operation on one array element

- Two simple kernels were selected for teaching the mapping of threads to data using threadIdx, blockIdx, blockDim, and gridDim variables
- The number of threads that we create is a multiple of the block dimension
- As a result, we will likely end up with more threads than data elements
- Not all threads will process elements of an array. We use an if statement to test if the global index values of a thread are within the valid range

Chapter 4:Data Parallel Execution Model

matrix multi is important

- MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL
- Matrix-matrix multiplication between an I * J matrix d_M and a J * K matrix d_N produces an I * K matrix d_P
- For simplicity, we will limit our discussion to square matrices, where I = J = K.
- We will use variable Width for I, J, and K.
- When performing a matrix-matrix multiplication, each element of the product matrix d_P is an inner product of a row of d_M and a column of d_N

Chapter 4:Data Parallel Execution Model

- MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

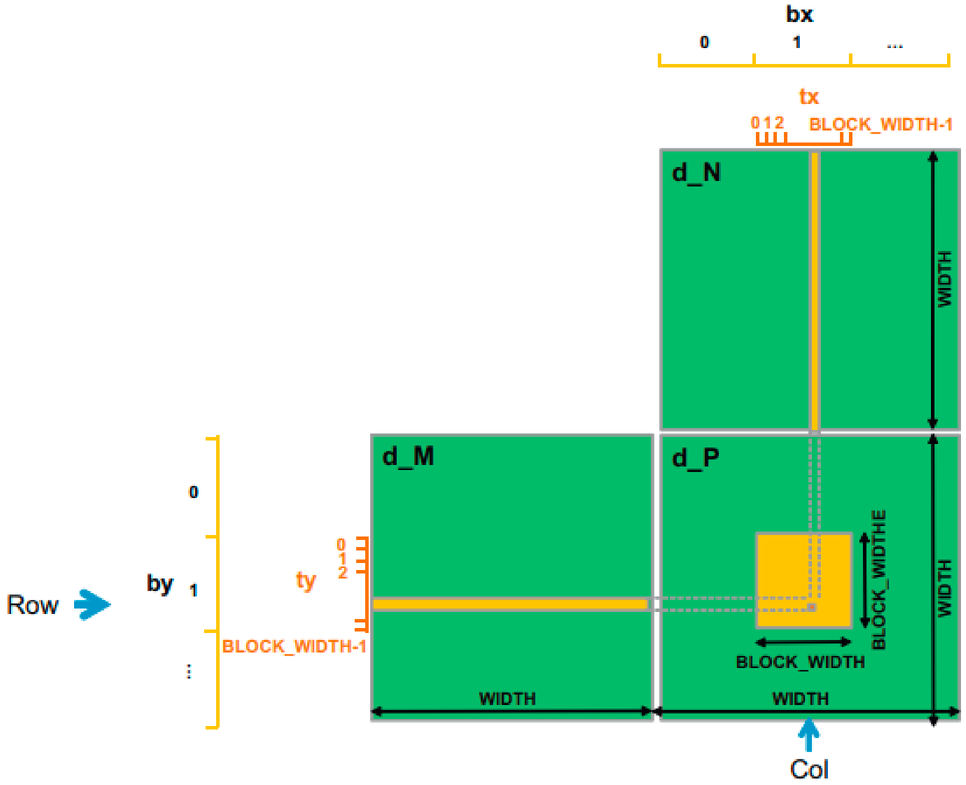


FIGURE 4.6

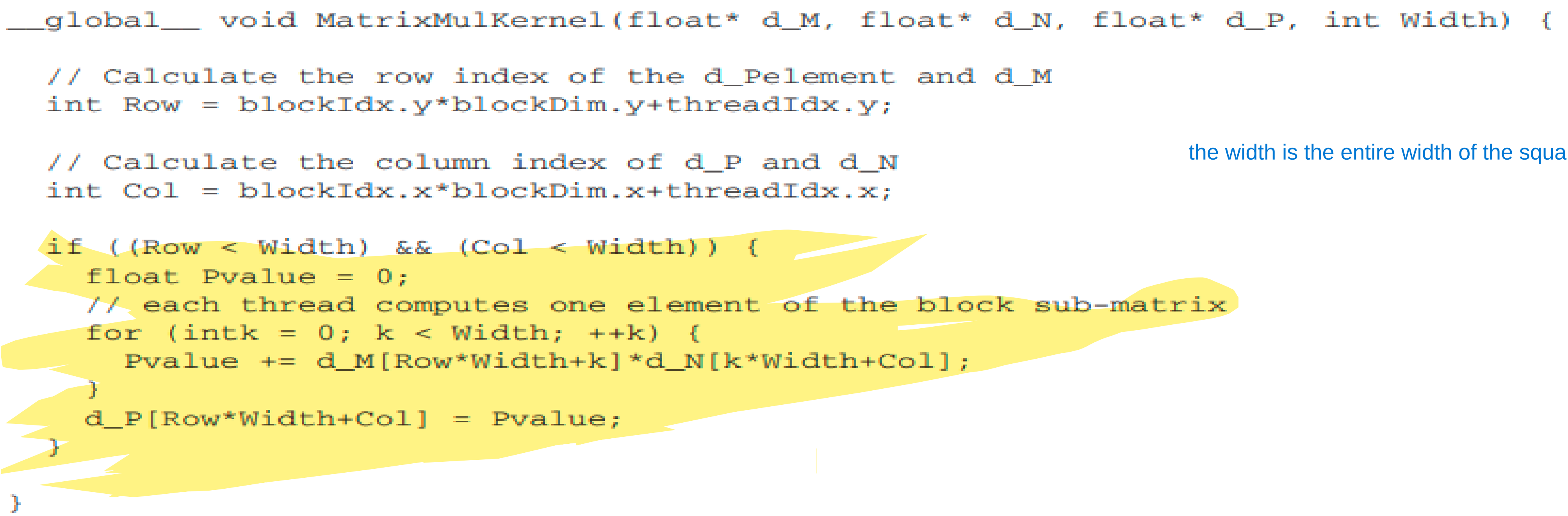
Matrix multiplication using multiple blocks by tiling d_P.

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- Each thread is responsible for calculating one d_P element
- The d_P element calculated by a thread is in
- Row: **blockIdx.y*blockDim.y + threadIdx.y**
- Column: **blockIdx. X*blockDim.x + threadIdx.x**

Chapter 4:Data Parallel Execution Model

- MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL



```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

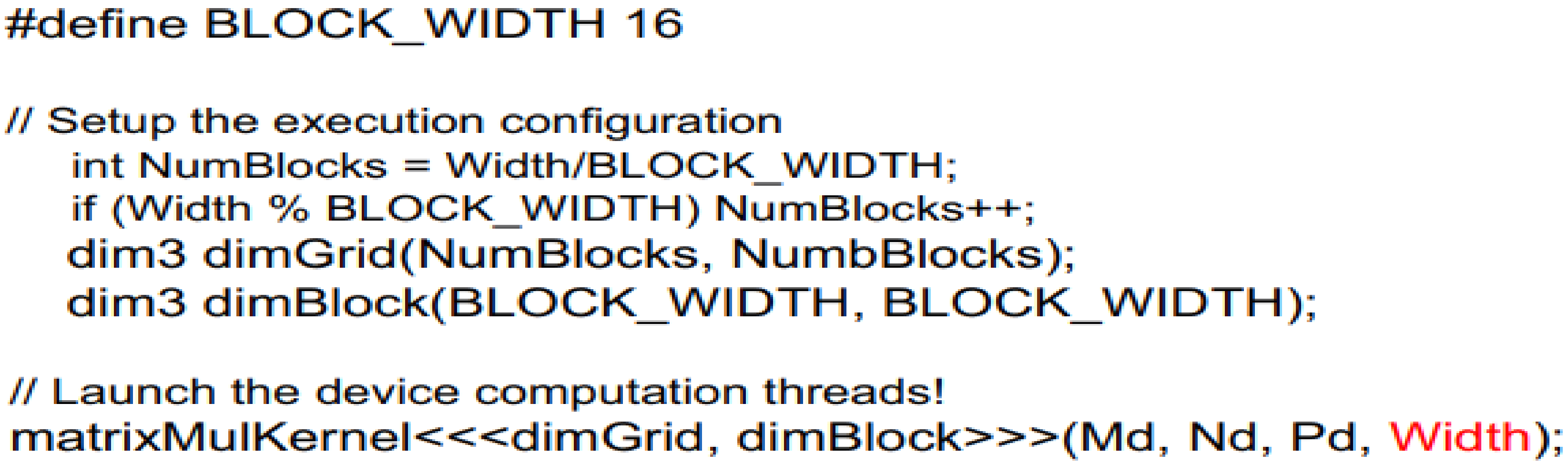
the width is the entire width of the square matrix

FIGURE 4.7

A simple matrix—matrix multiplication kernel using one thread to compute each d_P element.

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

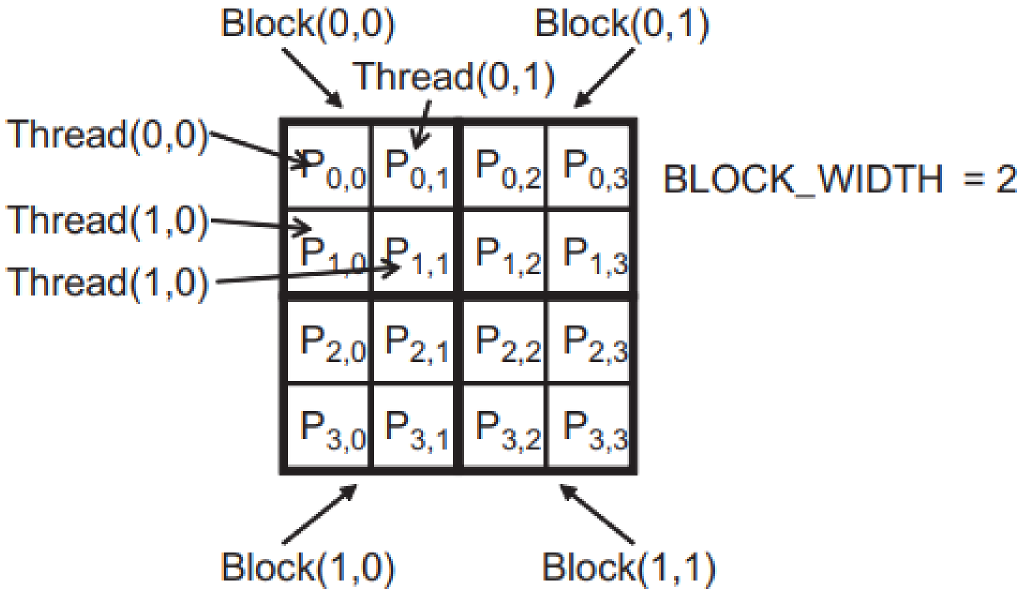
- Let us assume that, BLOCK_WIDTH=16
- Assume that we have a Width value of 1,000. That is, we need to do 1,000 x 1,000 matrix multiplication
- For a BLOCK_WIDTH value of 16, we will generate 16 x 16 blocks
- There will be 64 x 64 blocks in the grid to cover all d_P elements.



```
#define BLOCK_WIDTH 16

// Setup the execution configuration
    int NumBlocks = Width/BLOCK_WIDTH;
    if (Width % BLOCK_WIDTH) NumBlocks++;
   dim3 dimGrid(NumBlocks, NumbBlocks);
   dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads!
matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

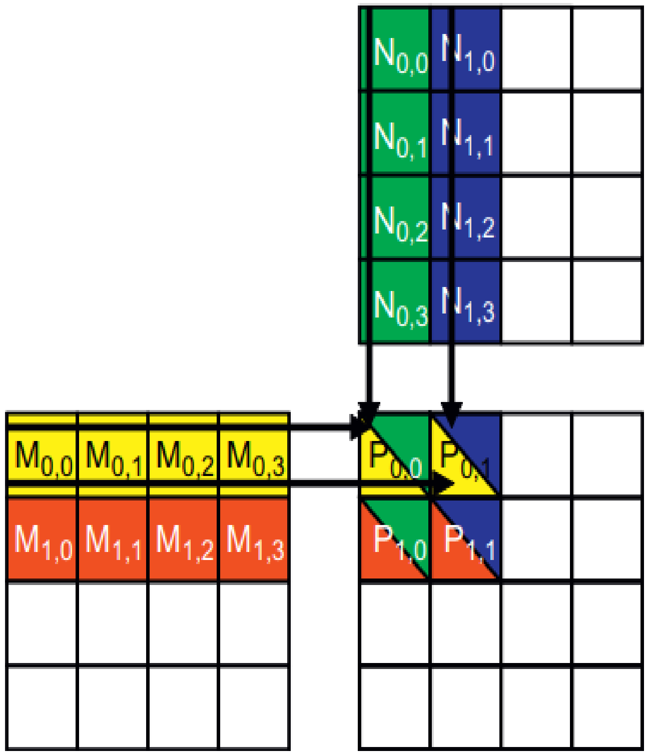
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- For a BLOCK_WIDTH=32, there will be 32 x 32 blocks in the grid to cover all d_P elements
- Let us consider a smaller matrix of size 4x4
- BLOCK_WIDTH = 2
- The d_P matrix is now divided into four tiles and each block calculates one tile


Block(0,0)
Block(0,1)
Thread(0,1)
Thread(0,0)
Thread(1,0)
Thread(1,0)
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$
BLOCK_WIDTH = 2
Block(1,0)
Block(1,1)

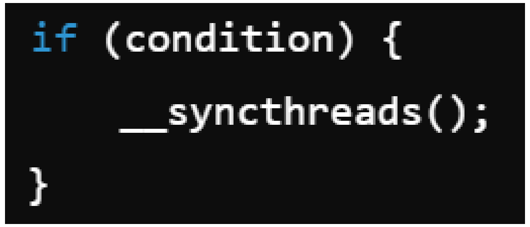
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- For the small matrix multiplication, threads in block(0,0) produce four dot products
- The Row and Col variables of thread(0,0) in block(0,0) are 0*0 + 0 = 0 and 0*0 + 0 =0
- It maps to P(0,0) and calculates the dot product of row 0 of M and column 0 of N



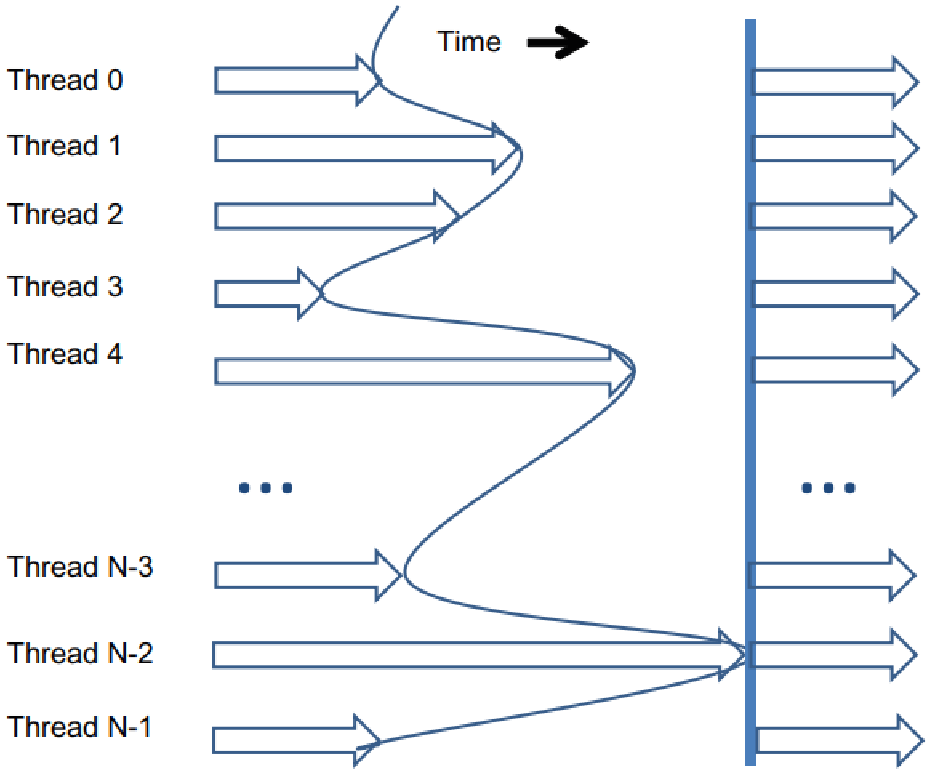
SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function **__syncthreads()**
- When a kernel function calls **__syncthreads(),** 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 of the kernel before any of them can move on to the next phase



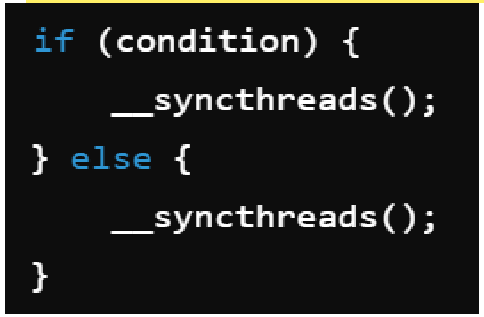
```
if (condition) {
    __syncthreads();
}
```

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

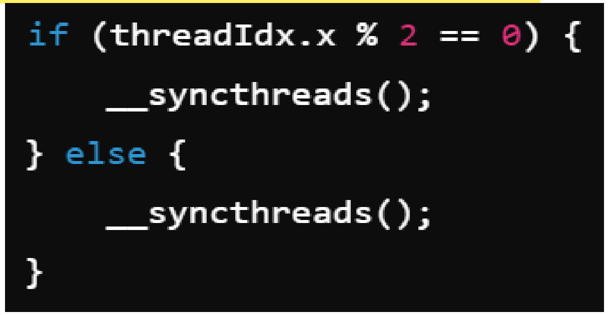


Synchronization and Transparent Scalability

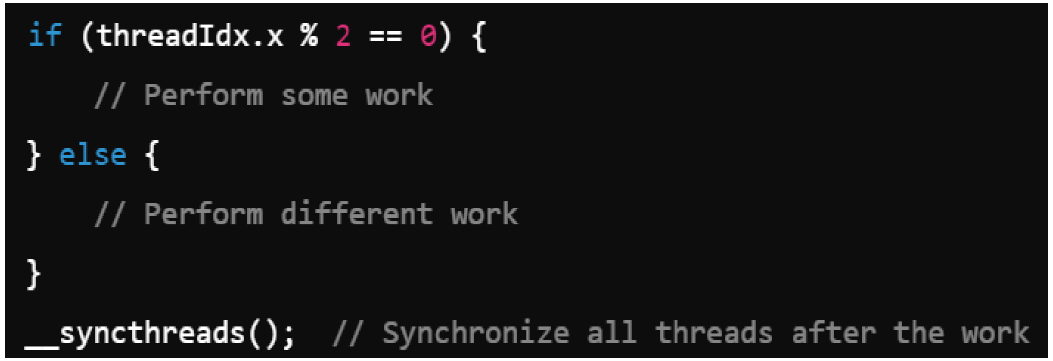
- In a typical if-then-else statement, different threads in a CUDA block can follow different execution paths based on a condition. However, __syncthreads() requires all threads in the block to reach the same synchronization barrier to ensure the program doesn't hang.
- For an **if-then-else** statement, 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.
- All threads in the block must evaluate condition the same way (either true or false), so that all threads execute the same __syncthreads() call.
- If some threads take the if path and others take the else path, the threads are synchronizing at different points, leading to deadlock. This happens because each group of threads is waiting for the others at a different barrier. They would end up waiting for each other forever.
- If you need to have different execution logic in if-else, you should place __syncthreads() outside of the conditional block, ensuring all threads eventually synchronize at the same point.



```
if (condition) {
    __syncthreads();
} else {
    __syncthreads();
}
```



```
if (threadIdx.x % 2 == 0) {
    __syncthreads();
} else {
    __syncthreads();
}
```



```
if (threadIdx.x % 2 == 0) {
    // Perform some work
} else {
    // Perform different work
}
__syncthreads();  // Synchronize all threads after the work
```

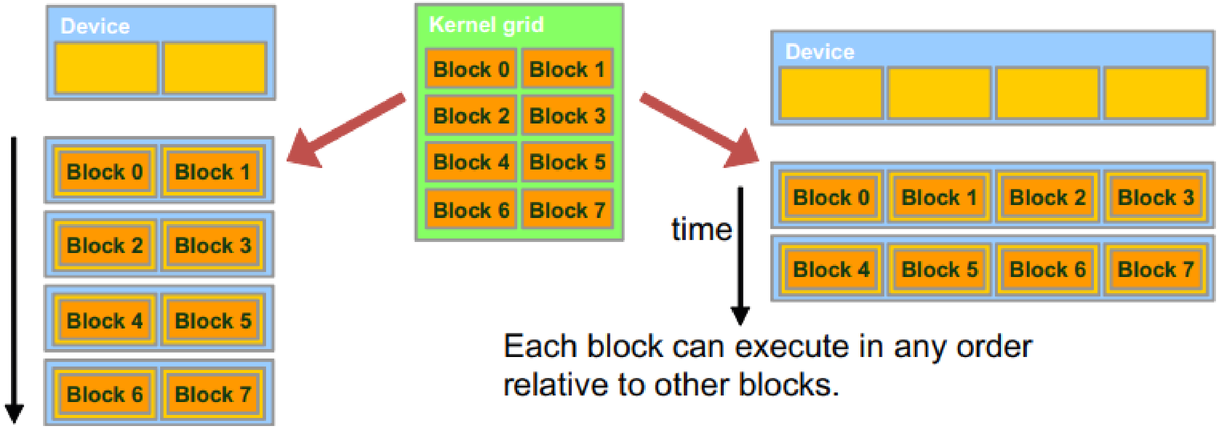
this is better and acceptable the other two not

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- One needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier
- Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever
- CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit
- A block can begin execution only when the runtime system has secured all the resources needed for all threads in the block to complete execution
- When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- Threads in different blocks cannot perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other since none of them need to wait for each other
- This flexibility enables scalable implementations



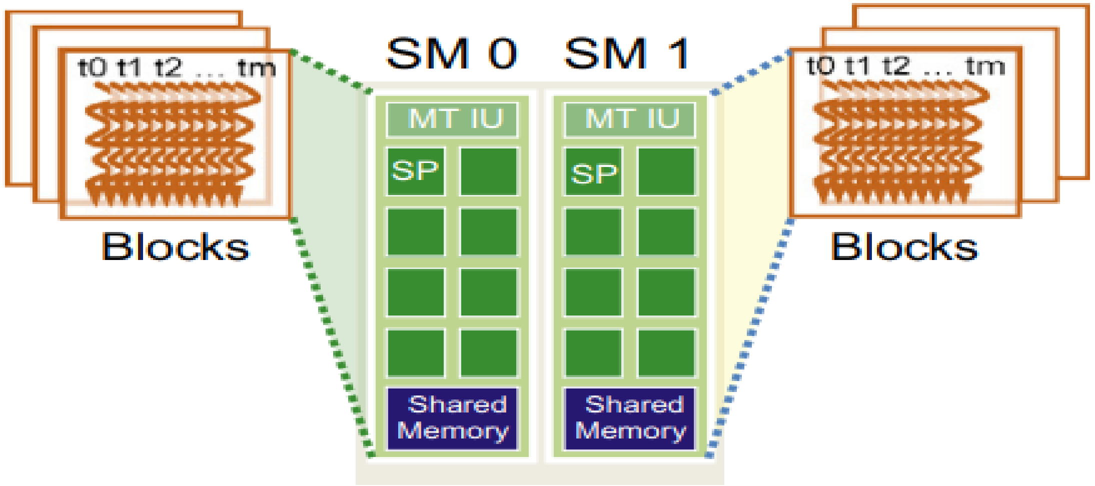
Each block can execute in any order relative to other blocks.

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- The ability to execute the same application code at a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of particular market segments
- Both execute exactly the same application program with no change to the code
- The ability to execute the same application code on hardware with a different number of execution resources is referred to as ***transparent scalability***, which reduces the burden on application developers and improves the usability of applications.

ASSIGNING RESOURCES TO BLOCKS

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads
- Threads are assigned to execution resources on a block-by-block basis
- In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs)



ASSIGNING RESOURCES TO BLOCKS

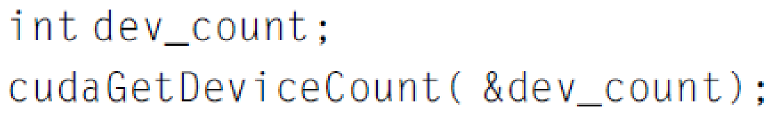
- For example, a CUDA device may allow up to eight blocks to be assigned to each SM
- In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of eight blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until their combined resource usage falls under the limit
- With a limited numbers of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device

ASSIGNING RESOURCES TO BLOCKS

- One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled
- It takes hardware resources for SMs to maintain the thread and block indices and track their execution status
- In more recent CUDA device designs, up to 1,536 threads can be assigned to each SM
- If a CUDA device has 30 SMs and each SM can accommodate up to 1,536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution

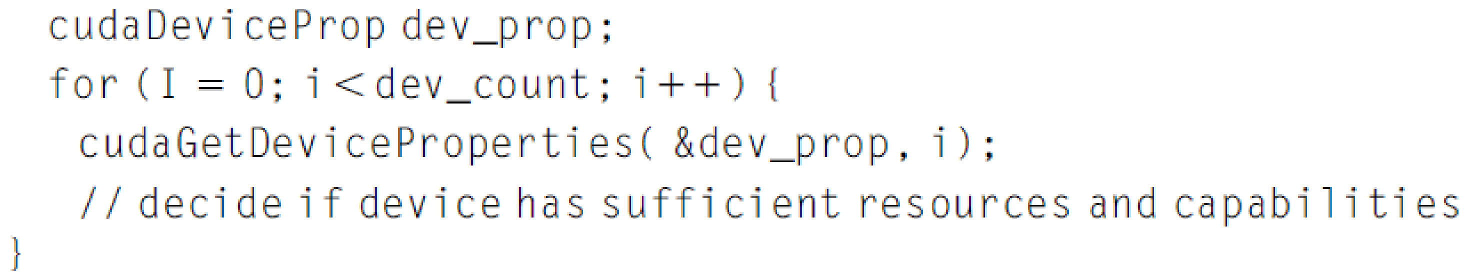
Querying Device Properties.

- In CUDA C, there is a built-in mechanism for host code to query the properties of the devices available in the system.
- The CUDA runtime system has an API function cudaGetDeviceCount() that returns the number of available CUDA devices in the system. The host code can find out the number of available CUDA devices using the following statements:



```
int dev_count;
cudaGetDeviceCount( &dev_count);
```

- Host code iterates through all the available devices, query their resources and capabilities, and choose the ones that have enough resources to execute the application with satisfactory performance.
- The CUDA runtime system numbers all the available devices in the system from 0 to dev_count-1. It provides an API function cudaGetDeviceProperties() that returns the properties of the device of which the number is given as an argument.
- For example, we can use the following statements in the host code to iterate through the available devices and query their properties:



```
 cudaDeviceProp dev_prop;
 for (I = 0; i<dev_count; i++) {
  cudaGetDeviceProperties( &dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

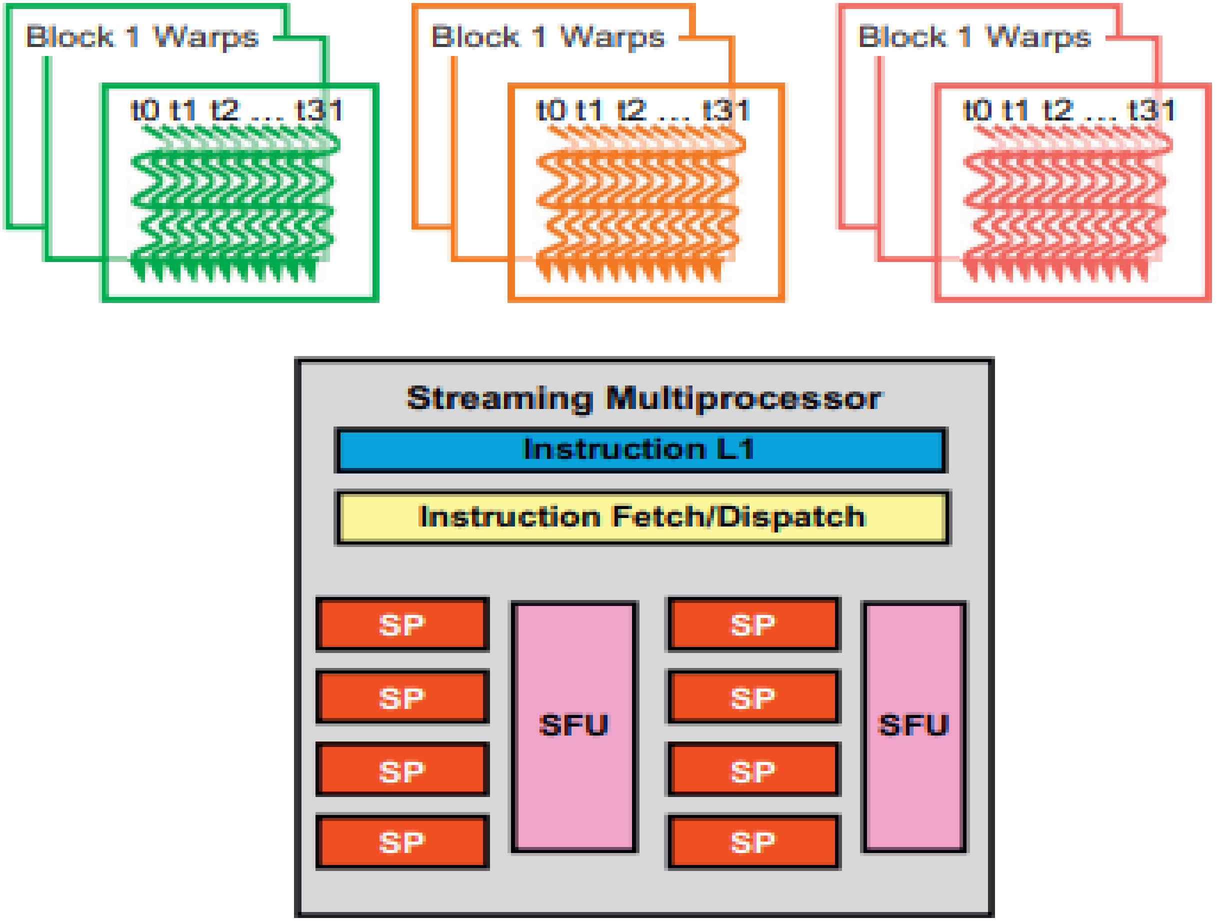
Querying Device Properties.

- **dev_prop.maxThreadsPerBlock:** gives the maximal number of threads allowed in a block in the queried device. Some devices allow upto 1,024 threads in each block and other devices allow fewer. Therefore, it is a good idea to query the available devices and determine which ones will allow a sufficient number of threads in each block as far as the application is concerned.
- **dev_prop.multiProcessorCount:** gives the number of SMs in the device. Some devices have only a small number of SMs (e.g., 2) and some have a much larger number of SMs (e.g., 30). If the application requires a large number of SMs to achieve satisfactory performance, it should definitely check this property of the prospective device.
- **dev_prop.clockRate:** gives the clock frequency of the device. The combination of the clock rate and the number of SMs gives a good indication of the hardware execution capacity of the device.
- The host code can find the maximal number of threads allowed along each dimension of a block in **dev_prop.maxThreadsDim[0]** (for the x dimension), **dev_prop.maxThreadsDim[1]** (for the y dimension), and **dev_prop.maxThreadsDim[2]** (for the z dimension).
- Similarly, it can find the maximal number of blocks allowed along each dimension of a grid in **dev_prop.maxGridSize[0]** (for the x dimension), **dev_prop.maxGridSize[1]** (for the y dimension), and **dev_prop.maxGridSize[2]** (for the z dimension).
- A typical use of this information is to determine whether a grid can have enough threads to handle the entire data set or if some kind of iteration is needed.

THREAD SCHEDULING AND LATENCY TOLERANCE

- Once a block is assigned to a SM, it is further divided into 32-thread units called **warps**
- The size of warps is implementation-specific
- In fact, warps are not part of the CUDA specification
- The size of warps is a property of a CUDA device, which is in the **dev_prop.warpSize**
- The warp is the unit of thread scheduling in SMs

THREAD SCHEDULING AND LATENCY TOLERANCE



THREAD SCHEDULING AND LATENCY TOLERANCE

- Each warp consists of 32 threads of consecutive **threadIdx** values: threads 0-31 form the first warp, 32-63 the second warp, and so on
- 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, we can determine that each block has 256/32 or 8 warps. With three blocks in each SM, we have 8 * 3 = 24 warps in each SM

THREAD SCHEDULING AND LATENCY TOLERANCE

- An SM is designed to execute all threads in a warp following the **single instruction, multiple data (SIMD)** model
- That is, at any instant in time, one instruction is fetched and executed for all threads in the warp
- These threads will apply the same instruction to different portions of the data
- As a result, all threads in a warp will always have the **same execution timing**

THREAD SCHEDULING AND LATENCY TOLERANCE

sp -> streaming processor it is cuda core ,sm will have multuple sp in it

- In general, there are fewer SPs than the number of threads assigned to each SM
- That is, each SM has only enough hardware to execute instructions from a small **subset of all threads** assigned to the SM at any point in time
- This is how CUDA processors efficiently execute **long-latency operations such as global memory accesses**

THREAD SCHEDULING AND LATENCY TOLERANCE

- When an instruction executed by the threads in a warp needs to 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 will be selected for execution
- If more than one warp is ready for execution, a **priority mechanism** is used to select one for execution
- This mechanism of filling the latency time of operations with work from other threads is often called **latency tolerance** or **latency hiding**

THREAD SCHEDULING AND LATENCY TOLERANCE

- Warp scheduling is also used for tolerating other types of operation latencies: **pipelined floating-point arithmetic** and **branch instructions**

- With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations

- 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**

THREAD SCHEDULING AND LATENCY TOLERANCE

- Assume that a CUDA device allows up to 8 blocks and 1,024 threads per SM, whichever becomes a limitation first. Furthermore, it allows up to 512 threads in each block. For matrix matrix multiplication, should we use 8 x 8, 16 x 16, or 32 x 32 thread blocks?

THREAD SCHEDULING AND LATENCY TOLERANCE

- If we use 8 x 8 blocks, each block would have only 64 threads. We will need 1,024/64 = 12 blocks to fully occupy an SM
- However, the limitation on number of blocks is 8
- Hence, at max, 8x64=512 threads in each SM (Under utilized)
- The 16 x 16 blocks give 256 threads per block. This means that each SM can take 1,024/256 = 4 blocks. This is within the 8-block limitation (best choice)
- The 32 x 32 blocks would give 1,024 threads in each block, exceeding the limit of 512 threads per block for this device

THREAD SCHEDULING AND LATENCY TOLERANCE

- If a CUDA device’s SM can take up to 1,536 threads and up to 4 thread blocks, which of the following block configurations would result in the most number of threads in the SM?
- A. 128 threads per block
- B. 256 threads per block
- C. 512 threads per block
- D. 1,024 threads per block

128 threads per block:

1. Maximum threads = 128×4=512 threads.
2. This configuration only utilizes 512 threads, which is far from the 1,536 maximum.

256 threads per block:

1. Maximum threads = 256×4=1,024 threads.
2. This configuration uses 1,024 threads, which is better but still doesn't fully utilize the SM.

512 threads per block:

1. Maximum threads = 512×3=1,536 threads.
2. This configuration uses exactly 1,536 threads, perfectly utilizing the SM’s limit.

1,024 threads per block:

1. Maximum threads = 1,024×1=1,0241 threads.
2. This configuration only uses 1,024 threads, but only one block is active, meaning underutilization of available blocks.

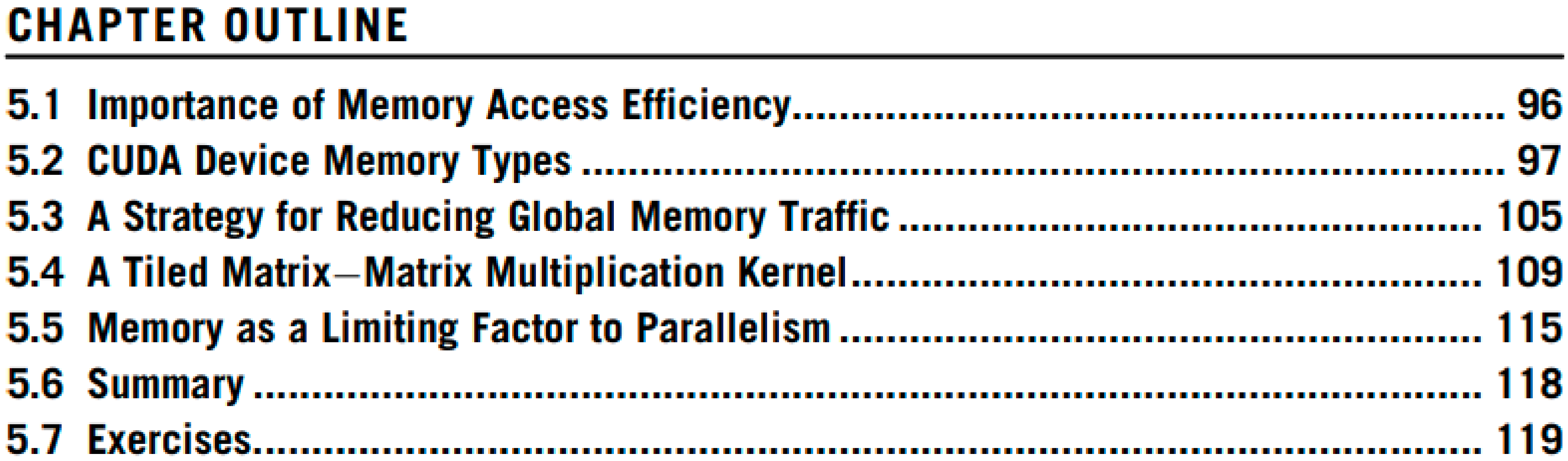
- **Answer:** The best configuration would be **C. 512 threads per block**, as it results in the maximum utilization of threads in the SM.

THREAD SCHEDULING AND LATENCY TOLERANCE

- For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?
- a. 2,000 b. 2,024 c. 2,048 d. 2,096

512*4=2048

Chapter 5: CUDA Memories

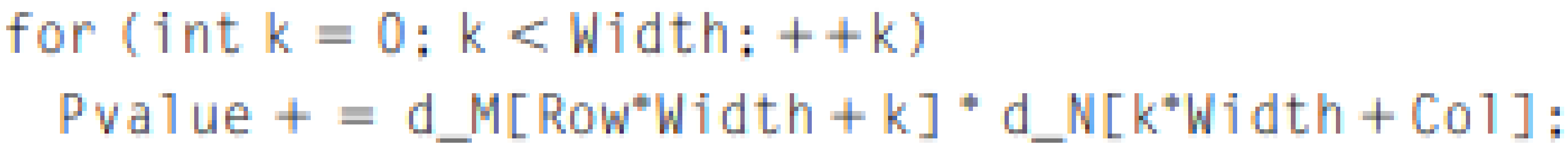


CHAPTER OUTLINE

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

latency -> delay for time it takes

- The global memory, which is typically implemented with dynamic random access memory (DRAM), tends to have **long access latencies** (hundreds of clock cycles) and **finite access bandwidth**
- Consider the example of matrix multiplication: The most important part of the kernel in terms of execution time is the for loop that performs inner product calculation



```
for (int k = 0; k < Width; ++k)
  Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
```

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition
- One global memory access fetches a d_M[] element and the other fetches a d_N[] element
- One floating-point operation multiplies the d_M[] and d_N[] elements fetched
- The other accumulates the product into Pvalue
- Thus, the ratio of floating-point calculation to global memory access operation is 1:1, or 1.0

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- This ratio is called as the **compute to global memory access (CGMA)** ratio
- Defined as the number of floating point calculations performed for each access to the global memory within a region of a CUDA program
- With a CGMA ratio of 1.0, the matrix multiplication kernel will execute no more than 50 giga floating-point operations per second
- We need to increase the CGMA ratio to achieve a higher level of performance for the kernel

CUDA DEVICE MEMORY TYPES

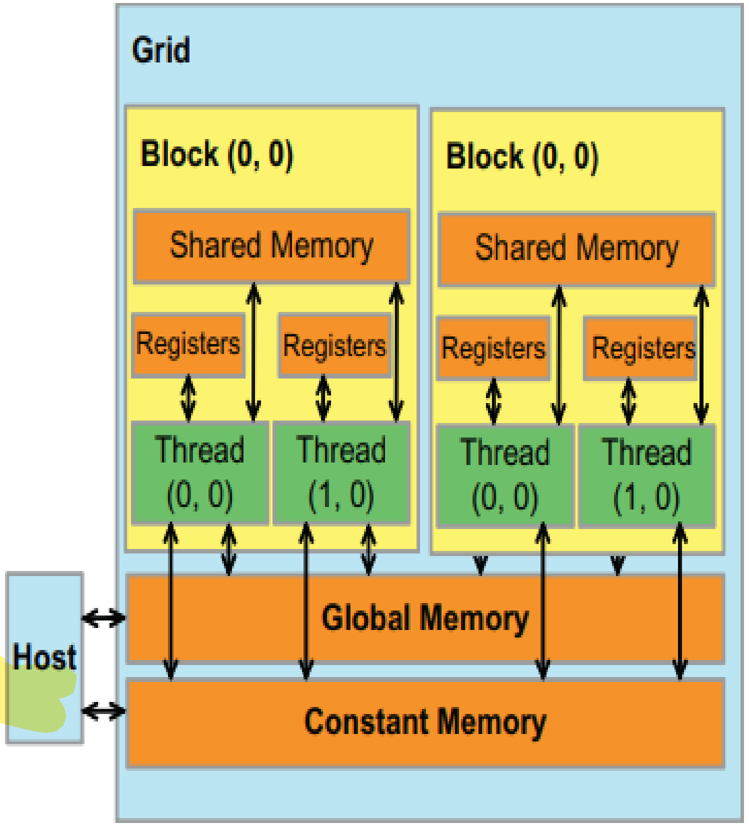
- Global memory and constant memory:
  - These types of memory can be written (W) and read (R) by the host by calling API functions
  - The constant memory supports short-latency, high-bandwidth, read-only access by the device
- Registers and shared memory:
  - On chip memories
  - Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner
  - Registers are allocated to individual threads; each thread can only access its own registers

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



CUDA DEVICE MEMORY TYPES

- Shared memory is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block

- Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work

- By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable

CUDA DEVICE MEMORY TYPES

- The global memory in the CUDA programming model maps to the memory of the von Neumann model
- The global memory is off the processor chip and is implemented with DRAM technology
  - Which implies **long access latencies** and relatively **low access bandwidth**
- The registers correspond to the "register file" of the von Neumann model
  - It is on the processor chip
  - Which implies very **short access latency** and **drastically higher access bandwidth**
  - Whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth
  - This will be reflected as an increase in the CGMA ratio

CUDA DEVICE MEMORY TYPES

- *Each access to registers involves fewer instructions than global memory*
- The processor uses the **PC** value to fetch instructions from memory into the **IR**
- The bits of the fetched instructions are then used to control the activities of the components of the computer
- Using the instruction bits to control the activities of the computer is referred to as **instruction execution**

In the context of computer architecture:

- **PC (Program Counter)**: This is a register in the CPU that holds the address of the next instruction to be executed. It keeps track of the sequence of instructions in a program.

- **IR (Instruction Register)**: This is a register that holds the current instruction being executed. Once the instruction is fetched from memory, it is placed in the IR for decoding and execution.

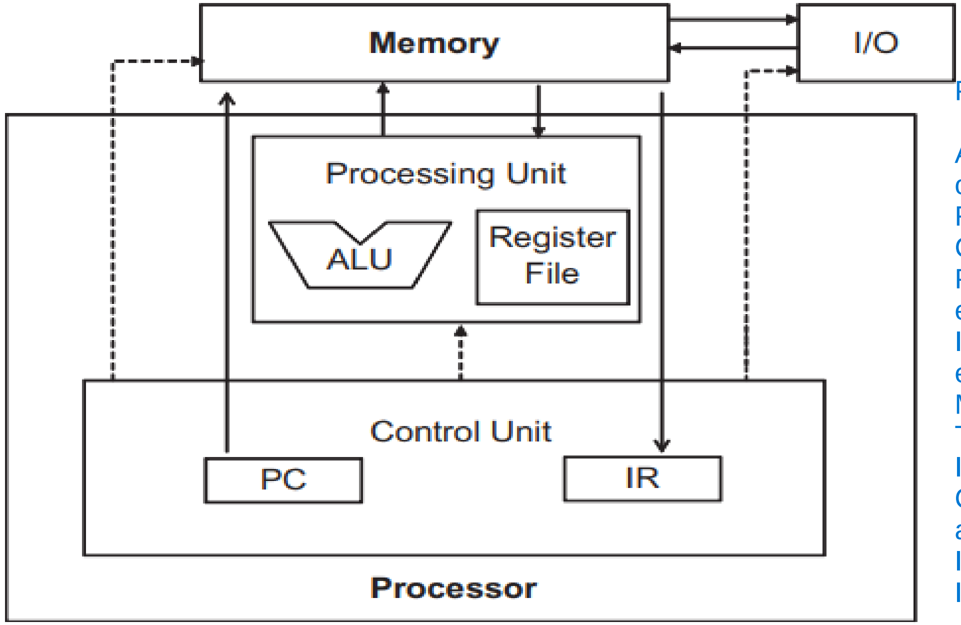
So, the statement "The processor uses the PC value to fetch instructions from memory into the IR" means that the CPU uses the address stored in the PC to locate and retrieve the next instruction from memory, which is then placed into the IR for processing. This is a fundamental part of the instruction cycle in a CPU.

CUDA DEVICE MEMORY TYPES

- The number of instructions that can be fetched and executed in each clock cycle is limited
- Therefore, the more instructions that need to be executed for a program, the more time it can take to execute the program

this image is that van model
here memory is global only
and register file is register memory

alu is arithmetic and logic unit



THE IMAGE SHOWS THIS Processor (CPU):

Processing Unit, which contains:

ALU (Arithmetic Logic Unit) - performs mathematical and logical operations
Register File - small, very fast memory locations within the CPU
Control Unit, which contains:
PC (Program Counter) - keeps track of which instruction to execute next
IR (Instruction Register) - holds the current instruction being executed
Memory:
This is the main system memory (RAM)
It's not showing local vs global memory like in GPUs
Connected to the processor via the system bus (shown by the arrows)
I/O:
Input/Output interface for connecting with external devices

CUDA DEVICE MEMORY TYPES

- Arithmetic instructions in most modern processors have “built-in” register operands

fadd r1, r2, r3

- Where **r2** and **r3** : the input operand values can be found
- The location for storing the floating-point addition result value is specified by **r1**
- when an operand of an arithmetic instruction is in a register, there is no additional instruction required to make the operand value available to the arithmetic and logic unit

Built-in Register Operands: Modern processors use registers as direct operands in instructions In the example fadd r1, r2, r3:fadd is floating-point additionr1, r2, and r3 are registersr2 and r3 contain the values to be added The result will be stored in r1

The Advantage: In older or simpler architectures, you might need multiple steps:

First load data from memory to registers

Then perform the operation Then store the result back With built-in register operands, it's all in one instruction

CUDA DEVICE MEMORY TYPES

- If an operand value is in global memory, one needs to perform a **memory load operation** to make the operand value available to the ALU
- If the first operand of a floating-point addition instruction is in global memory,

  load r2, r4, offset

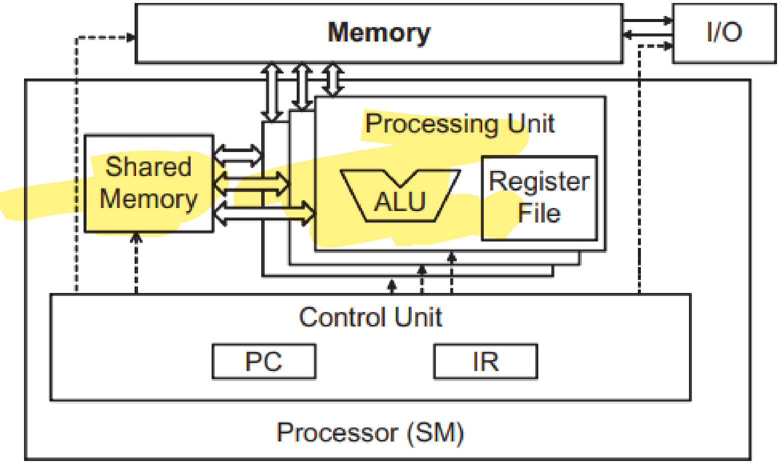
  fadd r1, r2, r3
- Since the processor can only fetch and execute a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without

CUDA DEVICE MEMORY TYPES

- In modern computers, the energy consumed for accessing a value from the register file is at least an order of magnitude lower than for accessing a value from the global memory

CUDA DEVICE MEMORY TYPES

- Shared memory is designed as part of the memory space that resides on the processor chip
- When the processor accesses data that resides in the shared memory, it needs to perform a memory load operation, just like accessing data in the global memory
- However, because shared memory resides on-chip, it can be accessed with much **lower latency and much higher bandwidth** than the global memory
- Because of the need to perform a load operation, shared memory has **longer latency and lower bandwidth** than registers
- In computer architecture, share memory is a form of scratchpad memory



CUDA DEVICE MEMORY TYPES

- Registers, shared memory, and global memory all have different functionalities, latencies, and bandwidth
- Each such declaration also gives its declared CUDA variable a **scope and lifetime**

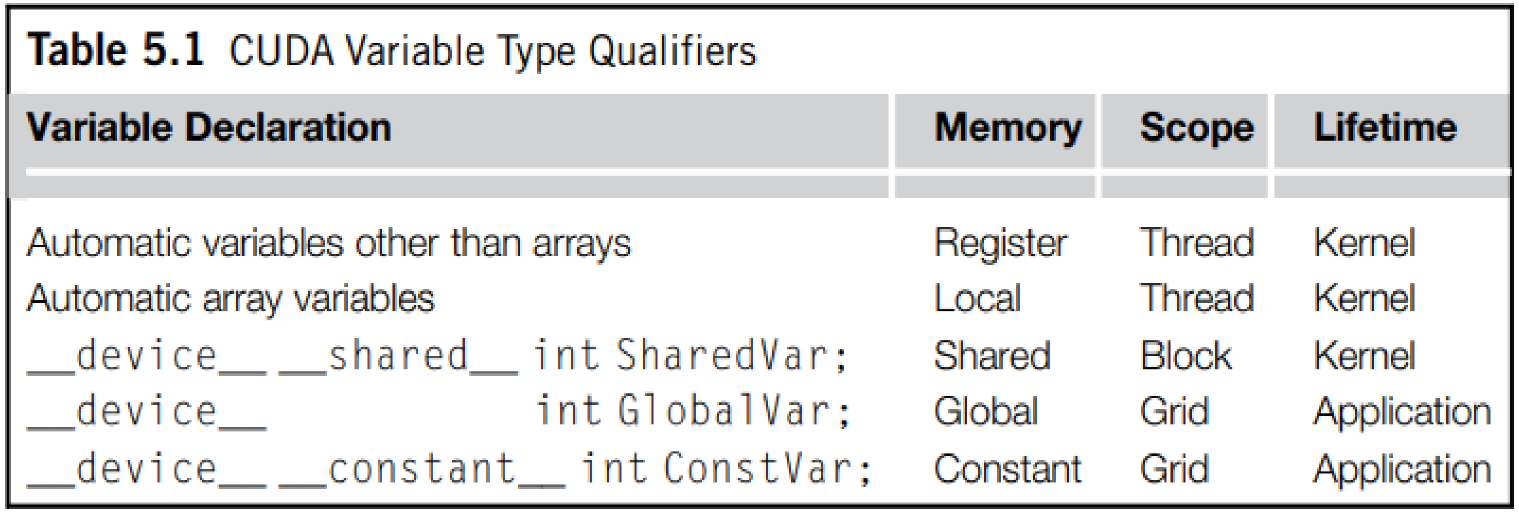


Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

CUDA DEVICE MEMORY TYPES

- Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids
- If a variable’s scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable
- For example, if a kernel declares a variable of which the scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable

CUDA DEVICE MEMORY TYPES

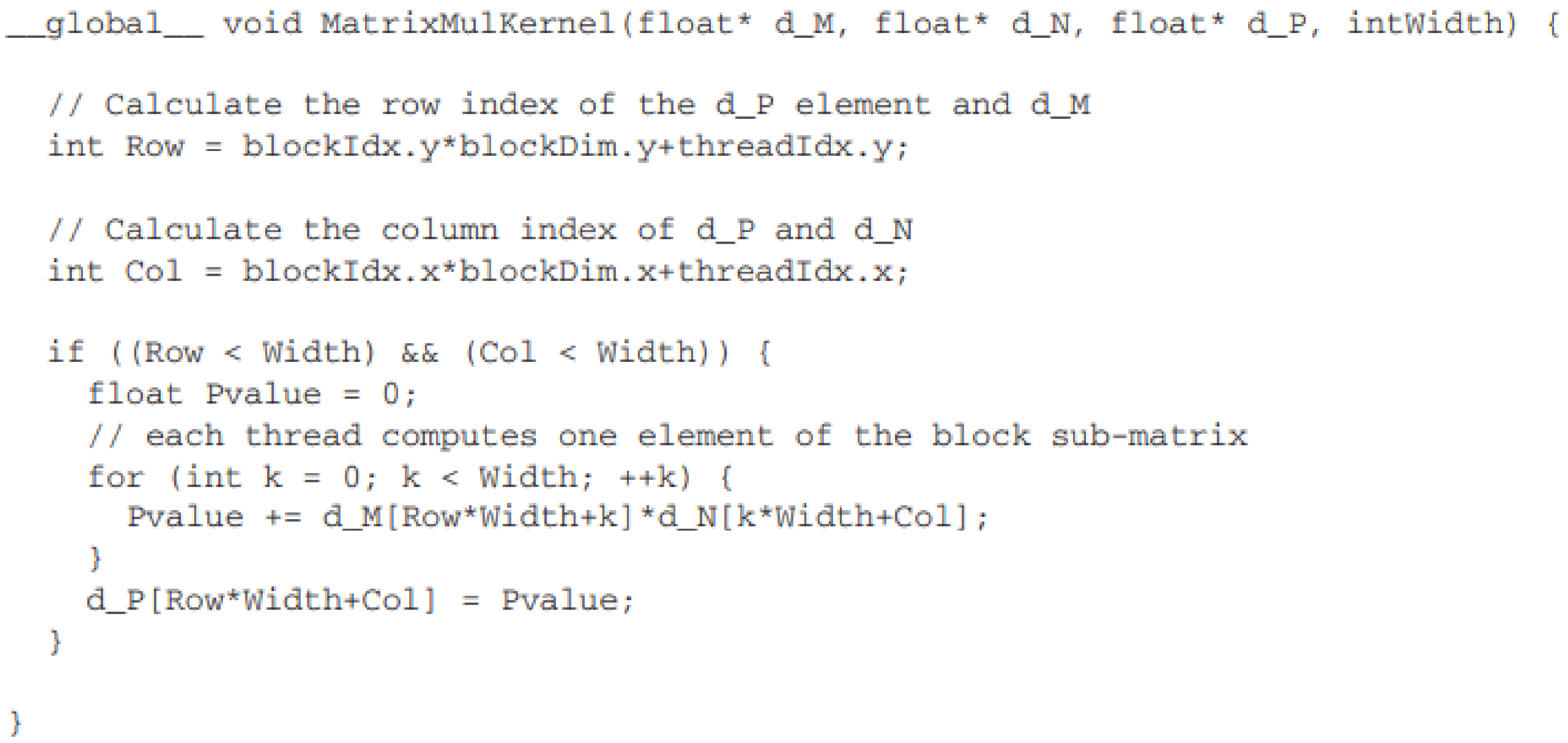
- Lifetime tells the portion of the program's execution duration when the variable is available for use:
  - Either within a **kernel's execution** or throughout the **entire application**
- If a variable's lifetime is within a kernel's execution, it must be declared within the kernel function body and will be available for use only by the kernel's code
- If the kernel is invoked several times, the value of the variable is not maintained across these invocations. Each invocation must initialize the variable to use them
- On the other hand, if a variable's lifetime is throughout the entire application, it must be declared outside of any function body
- The contents of these variables are maintained throughout the execution of the application and available to all kernels

CUDA DEVICE MEMORY TYPES

- All automatic **scalar variables** declared in kernel and device functions are placed into registers
- The scopes of these automatic variables are within **individual threads**
- When a kernel function declares an automatic variable, **a private copy of that variable is generated** for every thread that executes the kernel function
- When a thread terminates, all its automatic variables also cease to exist

CUDA DEVICE MEMORY TYPES

- Variables Row, Col, and Pvalue are all automatic variables



```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {

  // Calculate the row index of the d_P element and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

CUDA DEVICE MEMORY TYPES

- Automatic array variables are not stored in registers
- Instead, they are stored into the **global memory** and incur long access delays and potential access congestions
- The scope of these arrays is, like automatic scalar variables, **limited to individual threads**
- That is, a private version of each automatic array is created for and used by every thread
- Once a thread terminates its execution, the contents of its automatic array variables also cease to exist

CUDA DEVICE MEMORY TYPES

- If a variable declaration is preceded by the keyword **__shared__** it declares a shared variable in CUDA
- One can also add an optional **__device__** in front of **__shared__** in the declaration to achieve the same effect
- Such declaration typically resides within a kernel function or a device function
- Shared variables reside in shared memory. The scope of a shared variable is within a thread block
- A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel
- CUDA programmers often use shared variables to hold the portion of global memory data that are heavily used in an execution phase of a kernel

CUDA DEVICE MEMORY TYPES

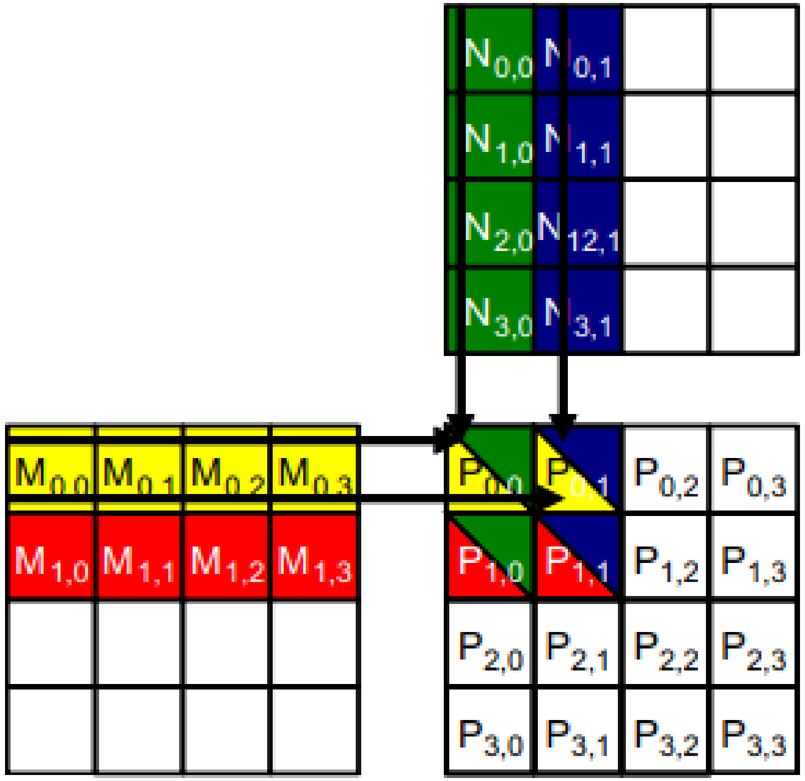
- If a variable declaration is preceded by the keyword **__constant__** it declares a constant variable in CUDA
- One can also add an optional **__device__** in front of **__constant__** to achieve the same effect
- Declaration of constant variables must be outside any function body
- The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution
- Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access

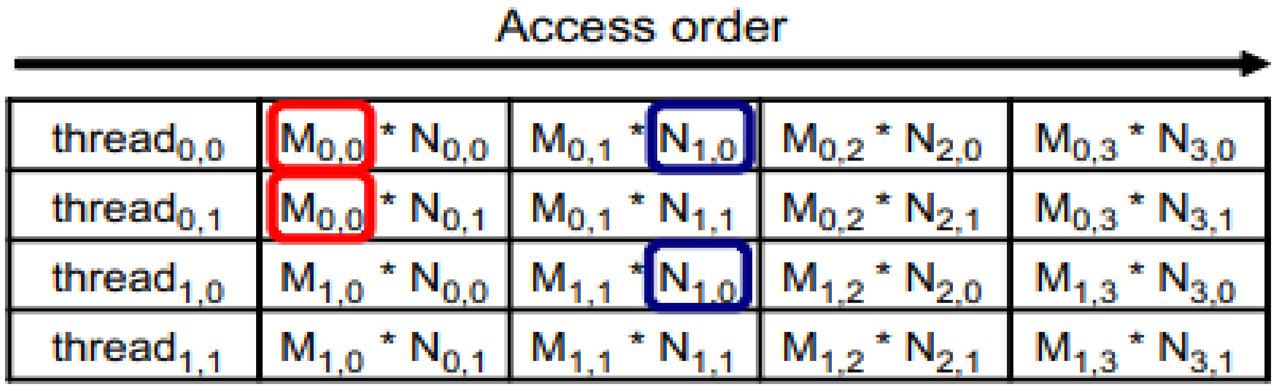
A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Trade-off in the use of device memories in CUDA:
  - Global memory is **large but slow**, whereas the shared memory is **small but fast**
- A common strategy is partition the data into subsets called **tiles** so that each tile fits into the shared memory
- An important criterion is that the kernel computation on these tiles can be done independently of each other

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Among the four threads highlighted, there is a significant overlap in terms of the M and N elements they access




Access order

$\text{thread}_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

considering 2x2 block so

- Every M and N element is accessed exactly twice during the execution of block0,0
- Therefore, if we can have all four threads to **collaborate in their accesses** to global memory, we can reduce the traffic to the global memory by half
- **Potential reduction** in global memory traffic in the matrix multiplication example is proportional to the **dimension of the blocks used**
- With N x N blocks, the potential reduction of global memory traffic would be N
- That is, if we use 16 x 16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads
- If multiple threads access data from the same DRAM location, they can form a “carpool” and combine their accesses into one DRAM request
- This, however, requires the threads to have a **similar execution schedule** so that their data accesses can be combined into one

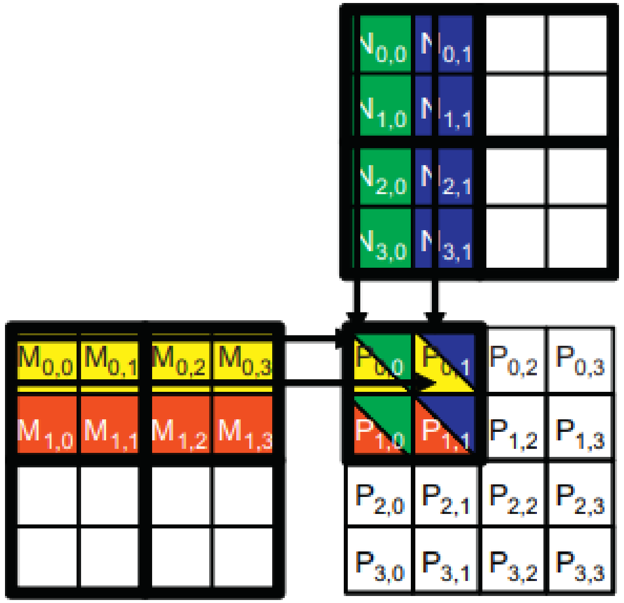
A TILED MATRIX MATRIX MULTIPLICATION KERNEL

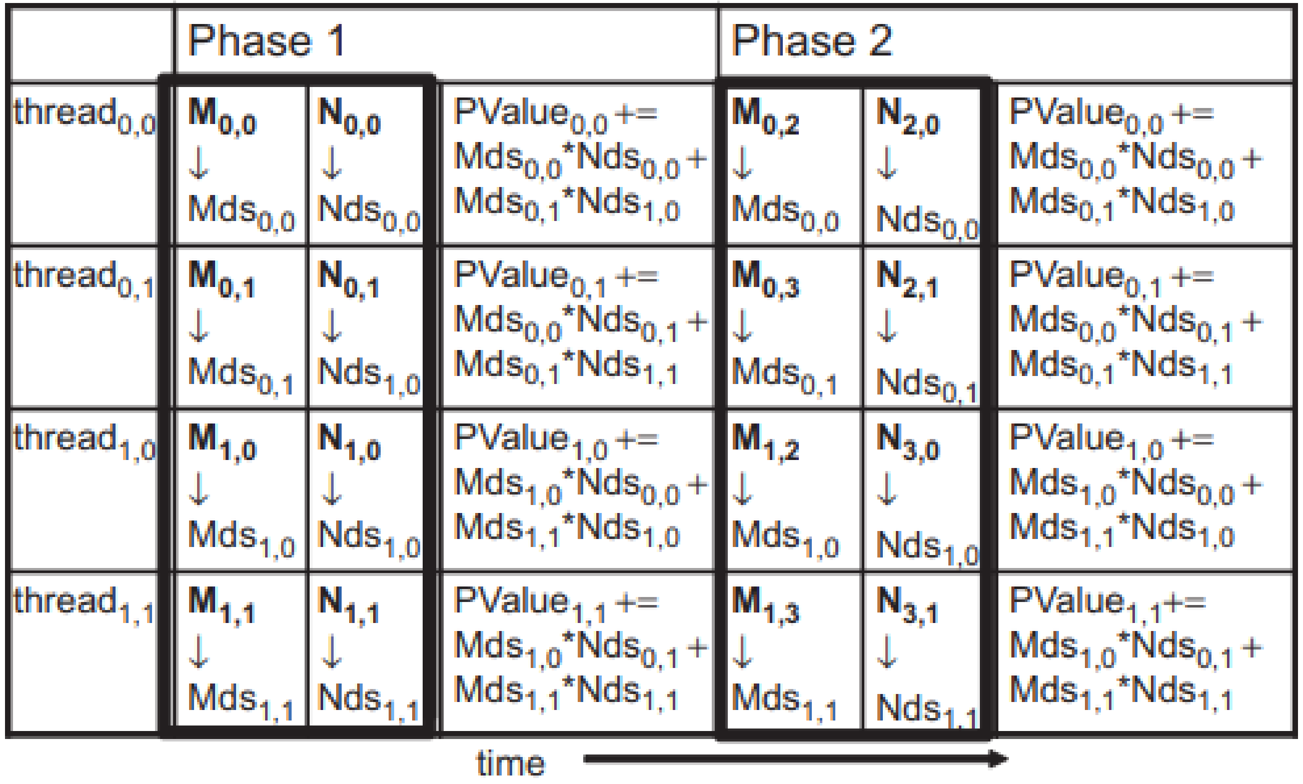
- The basic idea is to have the **threads to collaboratively** load M and N elements into the **shared memory** before they individually use these elements in their dot product calculation
- The size of the shared memory is **quite small** and one must be careful not to exceed the capacity of the shared memory when loading these M and N elements into the shared memory
- This can be accomplished by dividing the M and N matrices into **smaller tiles**
- The size of these tiles is chosen so that they can fit into the shared memory. In the simplest form, the tile dimensions equal those of the block

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- We divide the M and N matrices into 2 x 2 tiles
- The dot product calculations performed by each thread are now divided into phases
- In each phase, all threads in a block collaborate to load a tile of M elements and a tile of N elements into the shared memory
- This is done by having every thread in a block to load one M element and one N element into the shared memory

A TILED MATRIX MATRIX MULTIPLICATION KERNEL





	Phase 1			Phase 2		
$\text{thread}_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$\text{thread}_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$\text{thread}_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$\text{thread}_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- After the two tiles of M and N elements are loaded into the shared memory, these values are used in the calculation of the dot product
- Each value in the shared memory is used twice
- For example, the M1,1 value, loaded by thread1,1 into Mds1,1, is used twice, once by thread0,1 and once by thread1,1
- By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory
- In this case, we reduce the number of accesses to the global memory by half

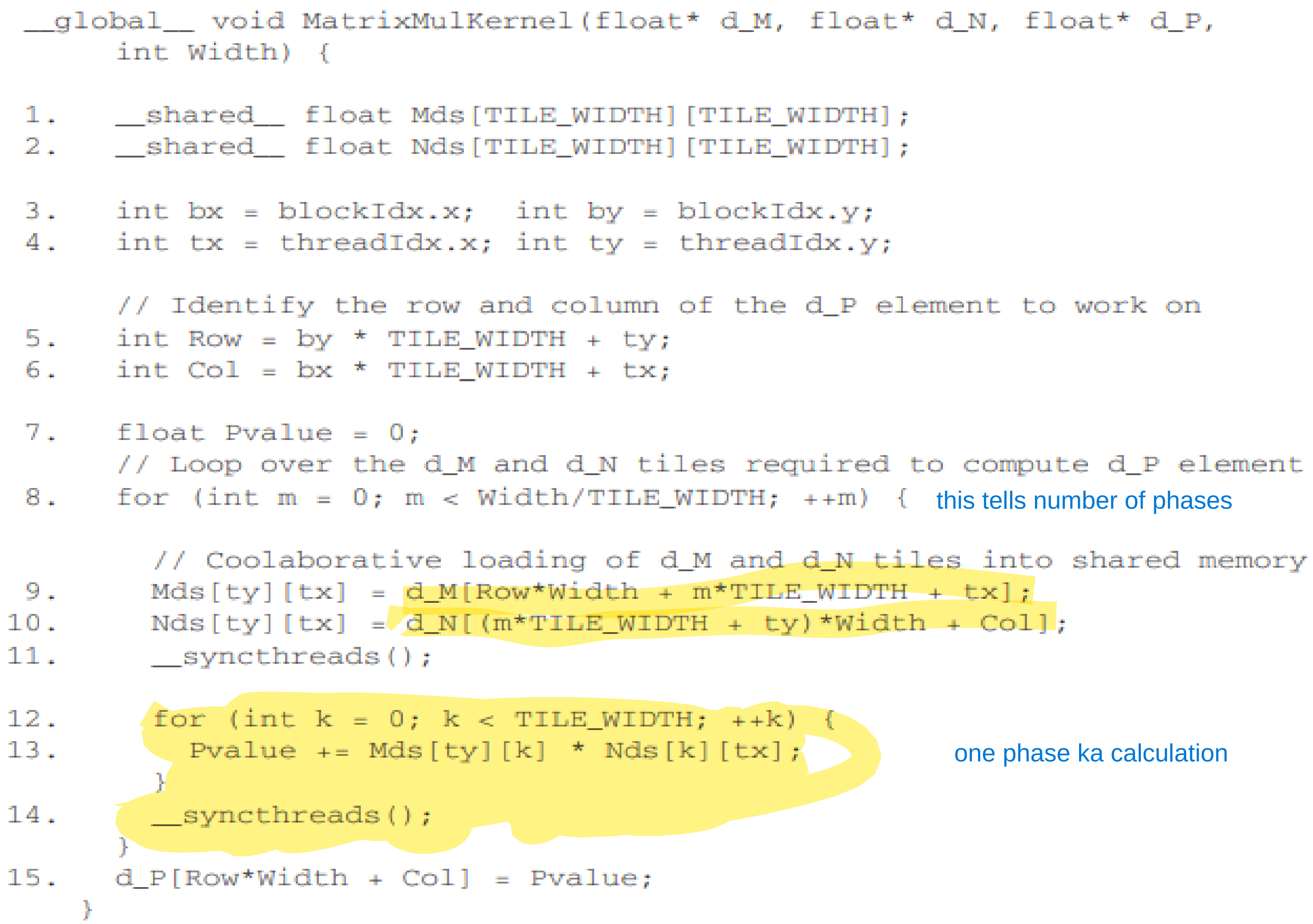
A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- If an input matrix is of dimension **N** and the tile size is **TILE_WIDTH**, the dot product would be performed in **N/TILE_WIDTH** phases
- The creation of these phases is key to the reduction of accesses to the global memory
- Note also that Mds and Nds are reused to hold the input values
- In each phase, the same Mds and Nds are used to hold the subset of M and N elements used in the phase
- This allows a much **smaller shared memory** to serve most of the accesses to global memory
- This is due to the fact that each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called **locality**

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- Locality, there is an opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory
- Locality is as important for achieving high performance in multicore CPUs as in many-thread GPUs

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

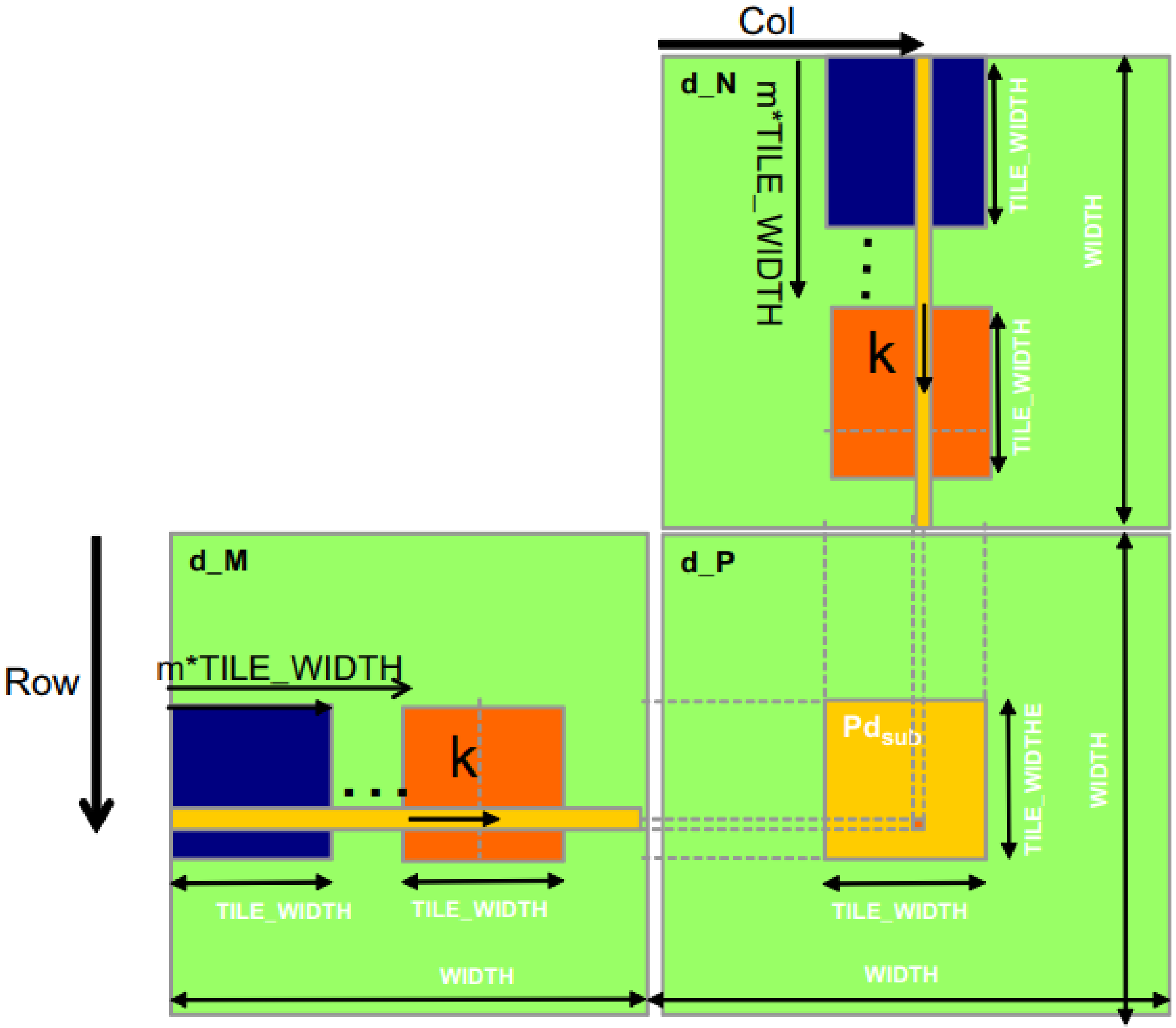


```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
      int Width) {

      __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;

      // Identify the row and column of the d_P element to work on
      int Row = by * TILE_WIDTH + ty;
      int Col = bx * TILE_WIDTH + tx;

      float Pvalue = 0;
      // Loop over the d_M and d_N tiles required to compute d_P element
      for (int m = 0; m < Width/TILE_WIDTH; ++m) {

        // Coolaborative loading of d_M and d_N tiles into shared memory
        Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
        Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
          Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
      }
      d_P[Row*Width + Col] = Pvalue;
    }
```

this tells number of phases

one phase ka calculation

A TILED MATRIX MATRIX MULTIPLICATION KERNEL



MEMORY AS A LIMITING FACTOR TO PARALLELISM

- CUDA registers and shared memory can be extremely effective in reducing the number of accesses to global memory
  - Should not exceed the capacity of these memories
- These memories are forms of **resources** that are needed for thread execution
- Each CUDA device offers a limited amount of resources, which limits the number of threads that can simultaneously reside in the SM for a given application
- In general, the more resources each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can reside in the entire device

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- Assume that in a device D, each SM can accommodate up to 1,536 threads and has 16,384 registers
- To support 1,536 threads, each thread can use only 16,384/1,536 = 10 registers
- If each thread uses 11 registers, the number of threads able to be executed concurrently in each SM will be reduced
    - Such reduction is done at the **block granularity/block level**

MEMORY AS A LIMITING FACTOR TO PARALLELISM

if extra register then entire block we cant so reduction in block wise will happen

- If each block contains 512 threads, the reduction of threads will be done by reducing **512 threads at a time**
- Thus, the next lower number of threads from 1,536 would be 512, a one-third reduction of threads that can simultaneously reside in each SM
- The number of registers available to each SM varies from device to device
- An application can dynamically determine the number of registers available in each SM of the device used and choose a version of the kernel that uses the number of registers appropriate for the device
- This can be done by calling the **cudaGetDeviceProperties()**
- **dev_prop.regsPerBlock** gives the number of registers available in each SM

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- Shared memory usage can also limit the number of threads assigned to each SM

- Assume device D has 16,384 (16 K) bytes of shared memory in each SM
- Shared memory is used by blocks
- Assume that each SM can accommodate up to eight blocks
- To reach this maximum, each block must not use more than 2K bytes of shared memory
- If each block uses more than 2K bytes of memory, the number of blocks that can reside in each SM is such that the total amount of shared memory used by these blocks does not exceed 16 K bytes
- For example, if each block uses 5K bytes of shared memory, no more than three blocks can be assigned to each SM

Chapter 6: Performance Considerations

Performance Considerations: Warps And Thread Execution

- The execution speed of a CUDA kernel can vary greatly depending on the **resource constraints of the device** being used
- Launching a CUDA kernel generates a **grid of threads** that are organized as a **two-level hierarchy**
- At the top level, a grid consists of a 1D, 2D, or 3D **array of blocks**
- At the bottom level, each block, in turn, consists of a 1D, 2D, or 3D **array of threads**
- Blocks can execute in any order relative to each other, which allows for transparent scalability in parallel execution of CUDA kernels

Warps And Thread Execution

- Threads in a block can execute in any order with respect to each other
- **Barrier synchronizations** should be used whenever we want to ensure all threads have completed a common phase of their execution before any of them start the next phase
- The correctness of executing a kernel should not depend on the fact that certain threads will execute in synchrony with each other

- Each thread block is partitioned into **warps**

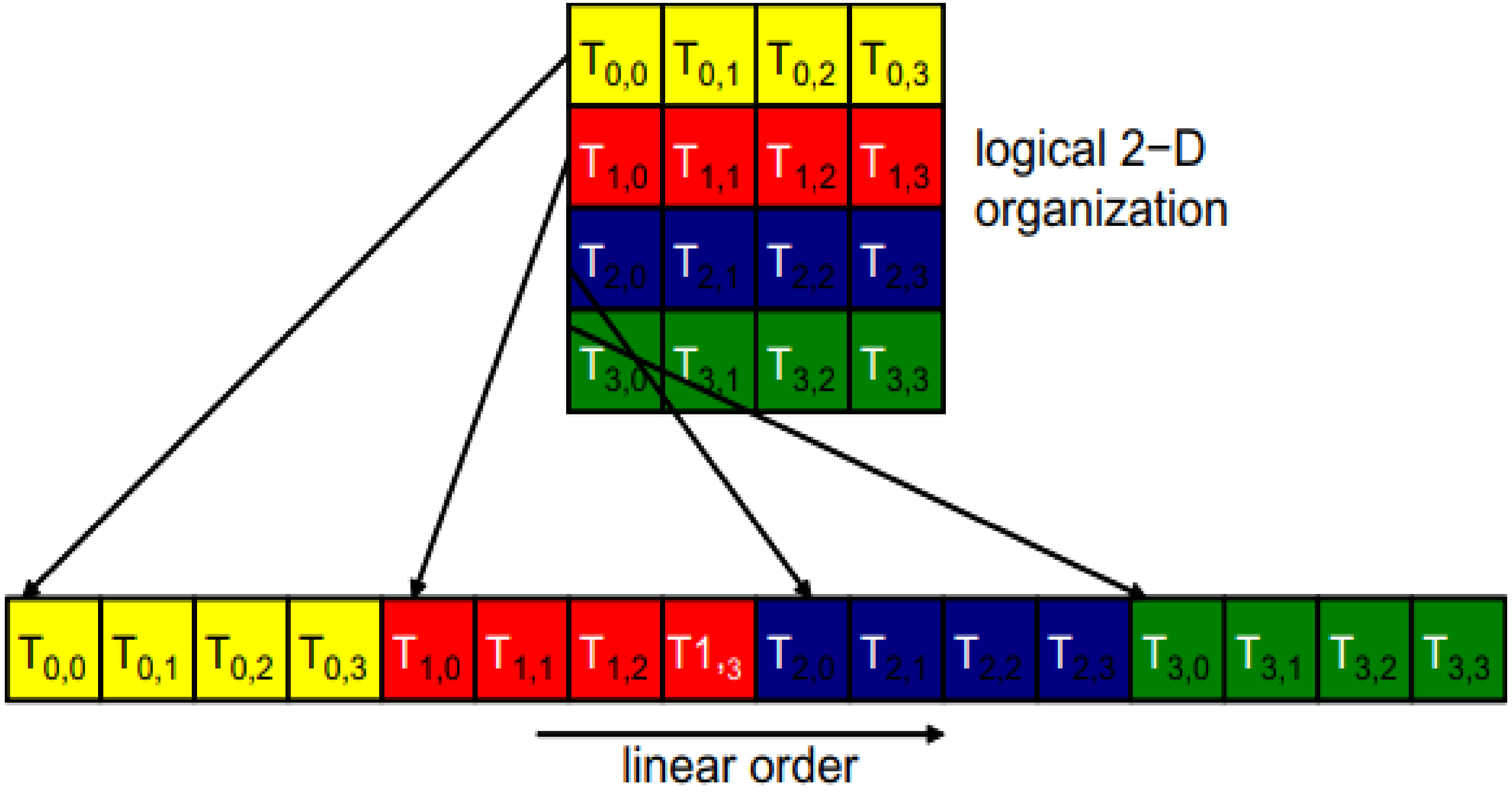
Warps And Thread Execution

- The size of a warp can easily vary from implementation to implementation
- Thread blocks are partitioned into warps based on **thread indices**
- If a thread block is organized into a 1D array (i.e., only threadIdx.x is used), the partition is straightforward; threadIdx.x values within a warp are consecutive and increasing
- For a warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63
- Warp n starts with thread **32 x n** and ends with thread **32(n + 1) - 1**

Warps And Thread Execution

- For a block of which the size is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads
- For example, if a block has 48 threads, it will be partitioned into two warps, and its warp 1 will be padded with 16 extra threads
- For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linear order before partitioning into warps

Warps And Thread Execution


$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2–D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T1_{,3}$ $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

Warps And Thread Execution

- For a 3D block, we first place all threads of which the threadIdx.z value is 0 into the linear order
- Among these threads, they are treated as a 2D block as shown in Figure
- All threads of which the threadIdx.z value is 1 will then be placed into the linear order, and so on
- The SIMD hardware executes all threads of a warp as a **bundle**
- **An instruction** is run for all threads in the same warp. It works well when all threads within a warp follow the same execution path, or control flow, when working their data

Warps And Thread Execution

- For example, for an **if-else construct**, the execution works well when either all threads execute the if part or all execute the else part
- When threads within a warp take different control flow paths, the SIMD hardware will take **multiple passes** through these divergent paths
- One pass executes those threads that follow the **if** part and another pass executes those that follow the **else** part
- During each pass, the threads that follow the other path are not allowed to take effect
- These passes are **sequential to each other**, thus they will add to the execution time

Warps And Thread Execution

- When threads in the same warp follow different paths of control flow, we say that these **threads diverge in their execution**
- The cost of divergence is the extra pass the hardware needs to take to allow the threads in a warp to make their own decisions
- Another example: Loops
- A control construct can result in thread divergence when its decision condition is based on threadIdx values
- For example, the statement **if (threadIdx.x > 2) {}**

Reduction algorithm

- A reduction algorithm derives a **single** value from an **array of values**
- The single value could be the sum, the maximal value, the minimal value, etc. among all elements
- A reduction can be easily done by sequentially going through every element of the array
- When an element is visited, the action to take depends on the type of **reduction** being performed
- For a **sum reduction**, the value of the element being visited at the current step, or the current value, is added to a running sum

Reduction algorithm

- For a **maximal reduction**, the current value is compared to a running maximal value of all the elements visited so far
- If the current value is larger than the running maximal, the current element value becomes the running maximal value
- For a **minimal reduction**, the value of the element currently being visited is compared to a running minimal
- If the current value is smaller than the running minimal, the current element value becomes the running minimal
- The sequential algorithm ends when all the elements are visited
- The sequential reduction algorithm is **work-efficient** in that every element is only visited once and only a minimal amount of work is performed when each element is visited
- Its execution time is proportional to the number of elements involved

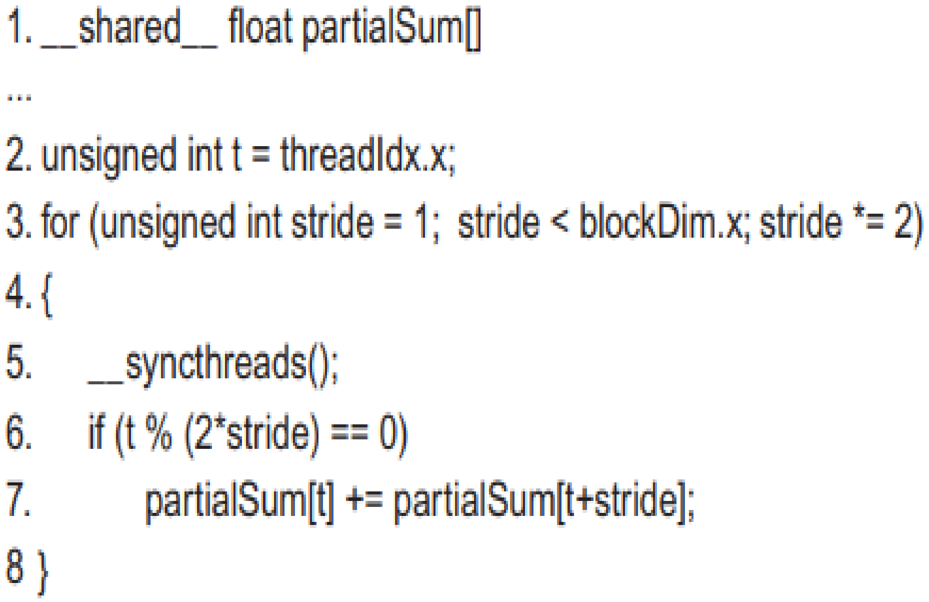
Reduction algorithms used in parallel programming

- Sequential Reduction: A basic reduction algorithm that processes elements one by one in a single thread.
- Tree-Based Parallel Reduction: Uses a tree structure to reduce pairs of elements in a parallel fashion.
- Pairwise Reduction: Elements are reduced in pairs, halving the number of active threads or processors after each pass.
- Warp-Level Intrinsics Reduction: An optimized reduction method within a warp (32 threads on NVIDIA GPUs) that avoids synchronization overhead.
- Segmented Reduction: Reduces elements within segments of an array independently. Useful for multidimensional or sparse data structures.

- Kogge-Stone Parallel Prefix-Sum: An efficient parallel scan algorithm that computes prefix sums and can be adapted for reduction.
- Reduction with Atomic Operations: Uses atomic operations to perform the reduction across multiple threads. Simple but can suffer from contention and performance issues.
- Hierarchical (Multi-Level) ReductionReduces data across multiple levels of parallelism (e.g., within threads, blocks, or nodes), common in GPU programming.
- Divide-and-Conquer Reduction: Recursively divides the input data, performs local reductions, and then combines the results in a divide-and-conquer style.

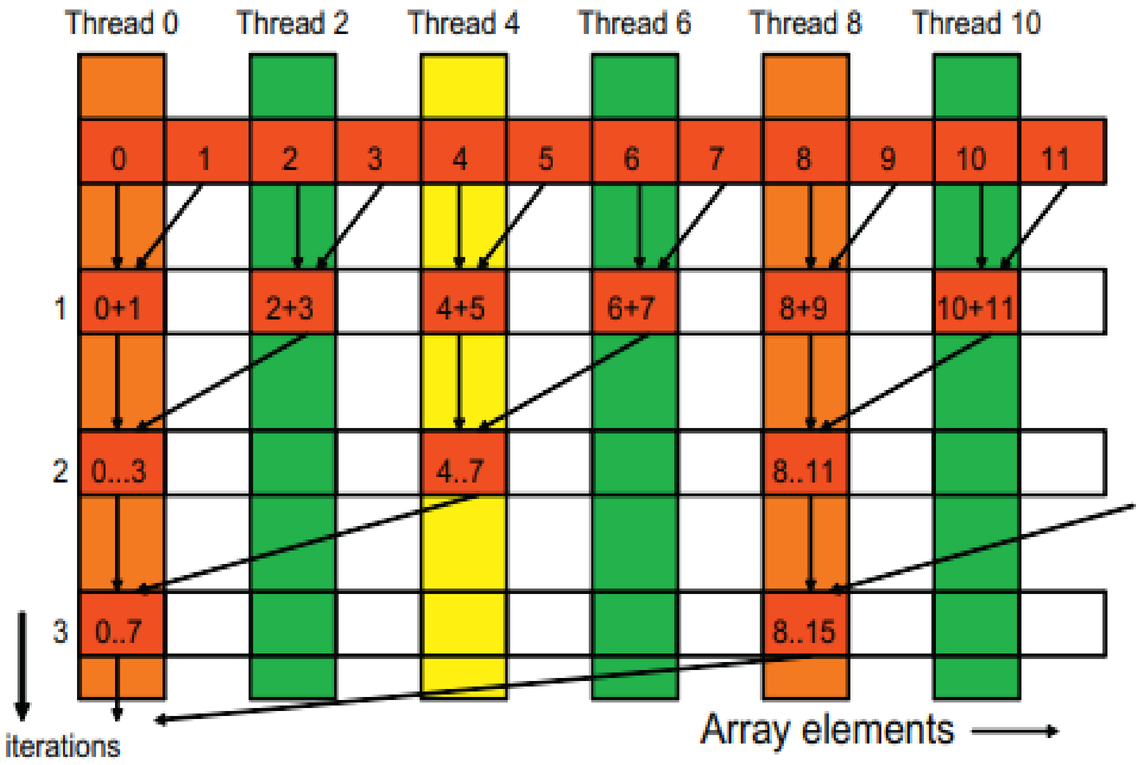
Reduction algorithm

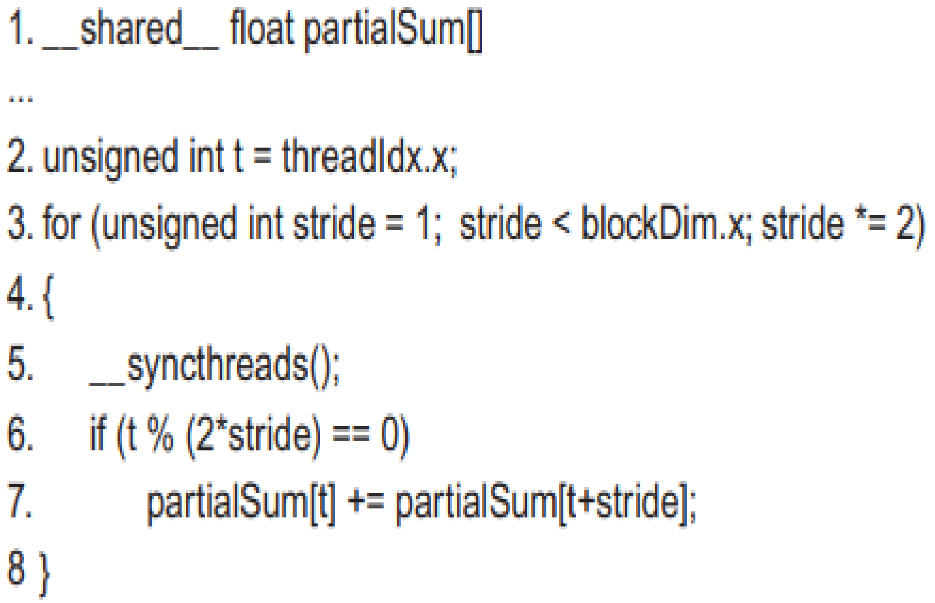
- The original array is in the **global memory**
- Each **thread block** reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction
- The reduction is done in place, which means the elements in the shared memory will be replaced by partial sums



```
__shared__ float partialSum[]
...
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;  stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
```

Reduction algorithm





```
__shared__ float partialSum[]
...
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;  stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
```

Reduction algorithm

- By using blockDim.x as the loop bound in line 4, the kernel assumes that it is launched with the same number of threads as the number of elements in the section
- Assume that the total number of elements to be reduced is N
  - The first round requires N/2 additions
  - The second round requires N/4 additions
  - The final round has only one addition
  - There are log2(N) rounds
  - The total number of additions performed by the kernel is

N/2 + N/4 + N/8 + ... +1 = N - 1

Reduction algorithm

- Therefore, the computational complexity of the reduction algorithm is O(N)
- During the first iteration of the loop, only those threads of which the threadIdx.x are even will execute the add statement
- One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute line 8
- In each successive iteration, fewer threads will execute line 8 but two passes will be still needed to execute all the threads during each iteration

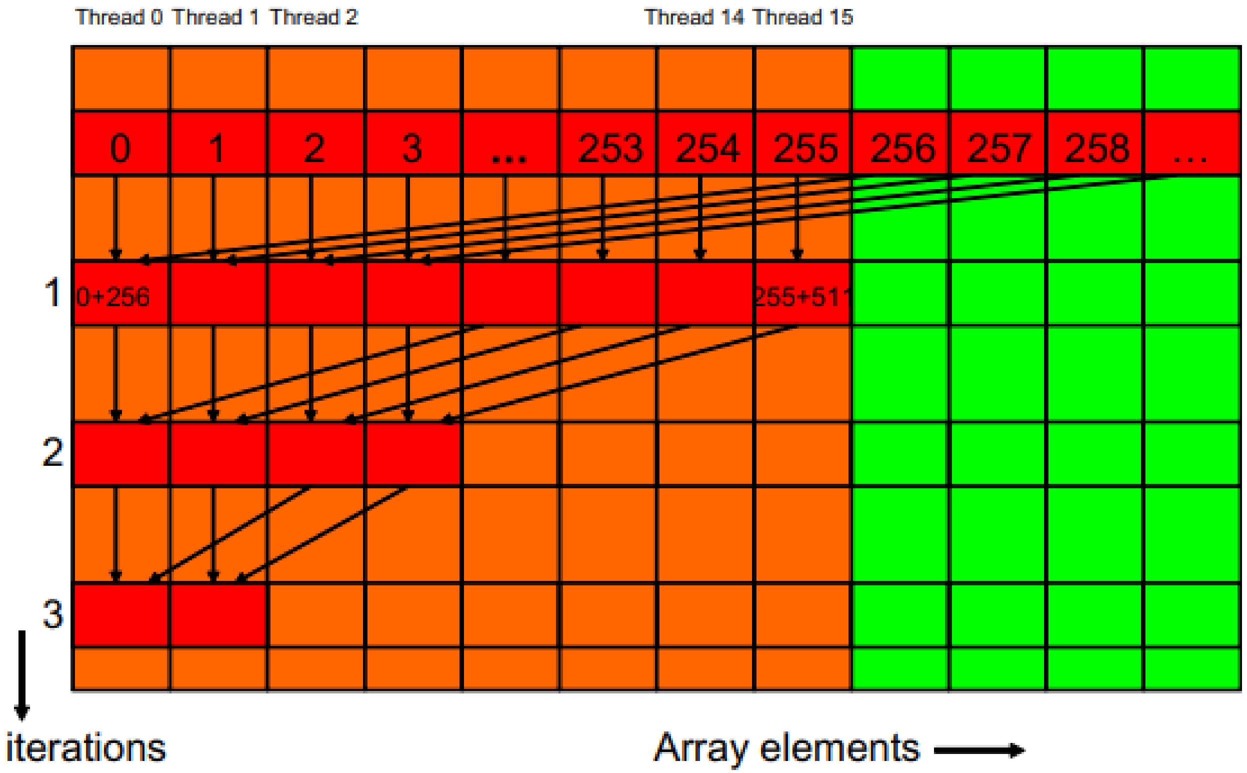
Pairwise Reduction algorithm

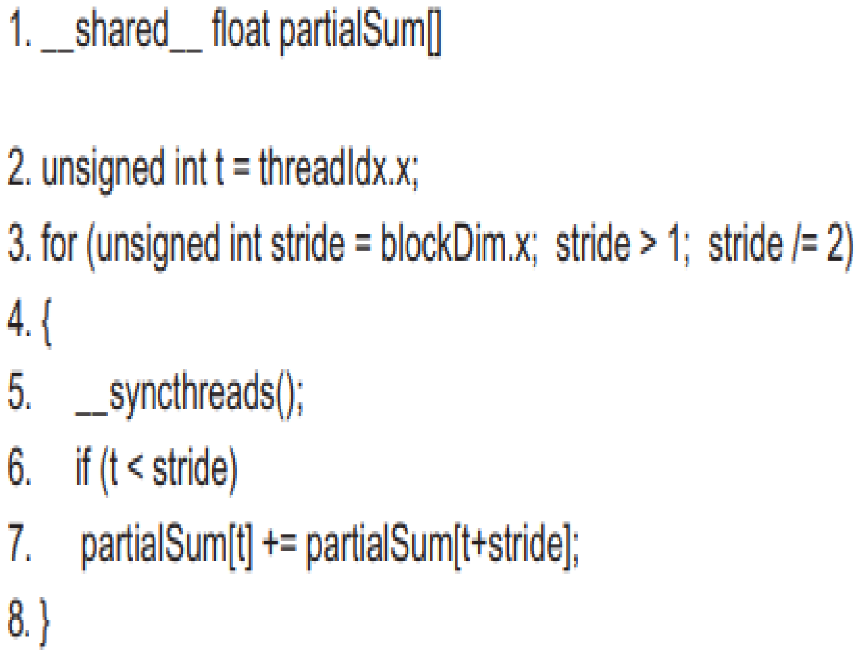
Example: Given an array [1, 2, 3, 4, 5, 6, 7, 8]

First Iteration (stride = 4):[1+5, 2+6, 3+7, 4+8] → [6, 8, 10, 12]

Second Iteration (stride = 2):[6+10, 8+12] → [16, 20]

Third Iteration (stride = 1):[16+20] → [36] (final sum)





```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;  stride > 1;  stride /= 2)
{
    __syncthreads();
    if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

Reduction algorithm

- Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other
- It does so by initializing the stride to be half the size of the section
- After the first iteration, all the pairwise sums are stored in the first half of the array
- The loop divides the stride by 2 before entering the next iteration

Reduction algorithm

- During the first iteration, all threads of which the threadIdx.x values are less than half of the size of the section execute line 7
- For a section of 512 elements, threads 0-255 execute the add statement during the first iteration
- While threads 255-511 do not
- The pairwise sums are stored in elements 0-255 after the first iteration
- Since the warps consist of 32 threads with consecutive threadIdx.x values, all threads in warps **0-7** execute the add statement, whereas warps **8-15** all skip the add statement
- **Since all threads in each warp take the same path, there is no thread divergence**

GLOBAL MEMORY BANDWIDTH

- One of the most important factors of CUDA kernel performance is accessing data in the global memory
- CUDA applications exploit **massive data parallelism**
- CUDA applications tend to process a massive amount of data from the global memory within a short period of time
- **Tiling techniques** utilizes shared memories to reduce the total amount of data that must be accessed by a collection of threads in the thread block
- **Memory coalescing** techniques that can more effectively move data from the global memory into shared memories and registers

GLOBAL MEMORY BANDWIDTH

- The global memory of a CUDA device is implemented with **DRAMs(Dynamic Random Access Memory)**
- Data bits are stored in **DRAM cells** that are **small capacitors**, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1
- The tiny amount of charge in the capacitor is shared with the bitline, which is highly capacitive. This is crucial because the bitline needs to be driven to a detectable voltage level.
- Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to be sensed by **sensor**
- It will set off its detection mechanism that determines whether a **sufficient amount of charge** is present in the capacitor to qualify as a “1”
- This process takes **tens of nanoseconds** in modern DRAM chips
- Because this is a very slow process relative to the desired data access speed (sub-nanosecond access per byte), modern DRAMs use parallelism to increase their rate of data access

- Activation and Row Access:
    - Row Addressing: When a read command is issued, the row containing the target cell is activated. This is done by sending the row address to the memory controller.
    - Row Decoder: The memory controller uses a row decoder to select the appropriate row, enabling access to the cells in that row.
- Charge Transfer:
    - Bitline Connection: The activated row connects its cells to a bitline (a highly capacitive wire) through the transistors.
    - Charge Sharing: The small charge in the capacitor of the targeted cell is shared with the bitline. Since the bitline has a larger capacitance, the charge from the capacitor causes a slight change in voltage on the bitline.
- Sensing Mechanism:
    - Sense Amplifier: A sense amplifier connected to the bitline detects this change in voltage.
    - Threshold Detection: The sense amplifier determines if the voltage exceeds a specific threshold. If it does, the cell is read as a "1"; if not, it’s a "0."

GLOBAL MEMORY BANDWIDTH

- Each time a DRAM location is accessed, many **consecutive locations** that include the requested location are actually accessed
- **Many sensors** are provided in each DRAM chip and they work in parallel
- Each senses the content of a bit within these consecutive locations
- Once detected by the sensors, the data from all these consecutive locations can be transferred at very high speed to the processor
- If an application can make **focused use of data from consecutive locations**, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed

GLOBAL MEMORY BANDWIDTH

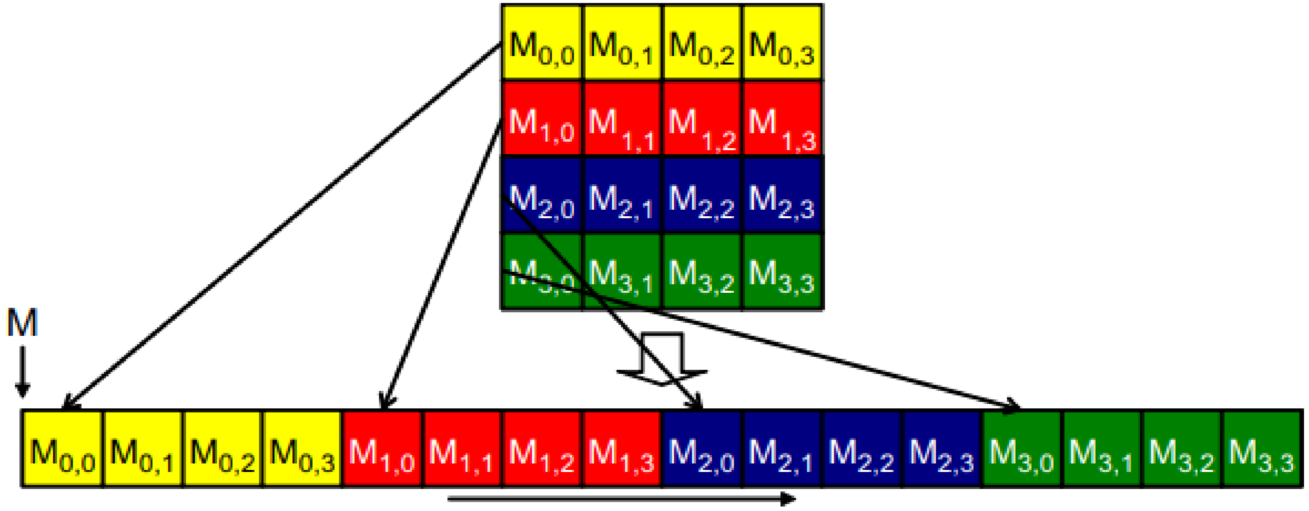
- Current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by **organizing memory accesses of threads into favorable patterns**
- **This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time**
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations
  - The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations
- In this case, the hardware **combines, or coalesces**, all these accesses into a consolidated access to consecutive DRAM locations

GLOBAL MEMORY BANDWIDTH

- For example, for a given load instruction of a warp, if thread 0 accesses global memory location N, thread 1 location N + 1, thread 2 location N + 2, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs
- Such coalesced access allows the DRAMs to deliver data at a rate close to the **peak global memory bandwidth**

GLOBAL MEMORY BANDWIDTH

- Multidimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the **row-major convention**



$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

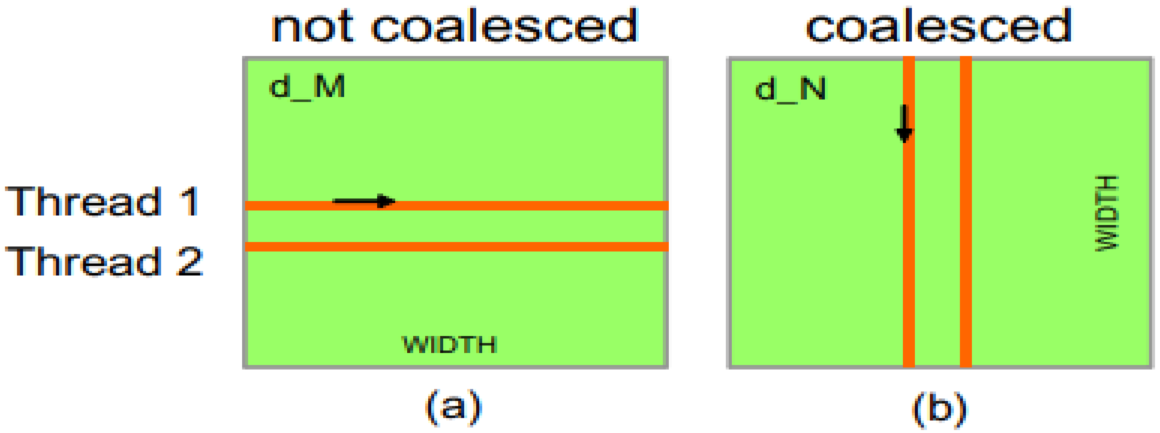
M

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

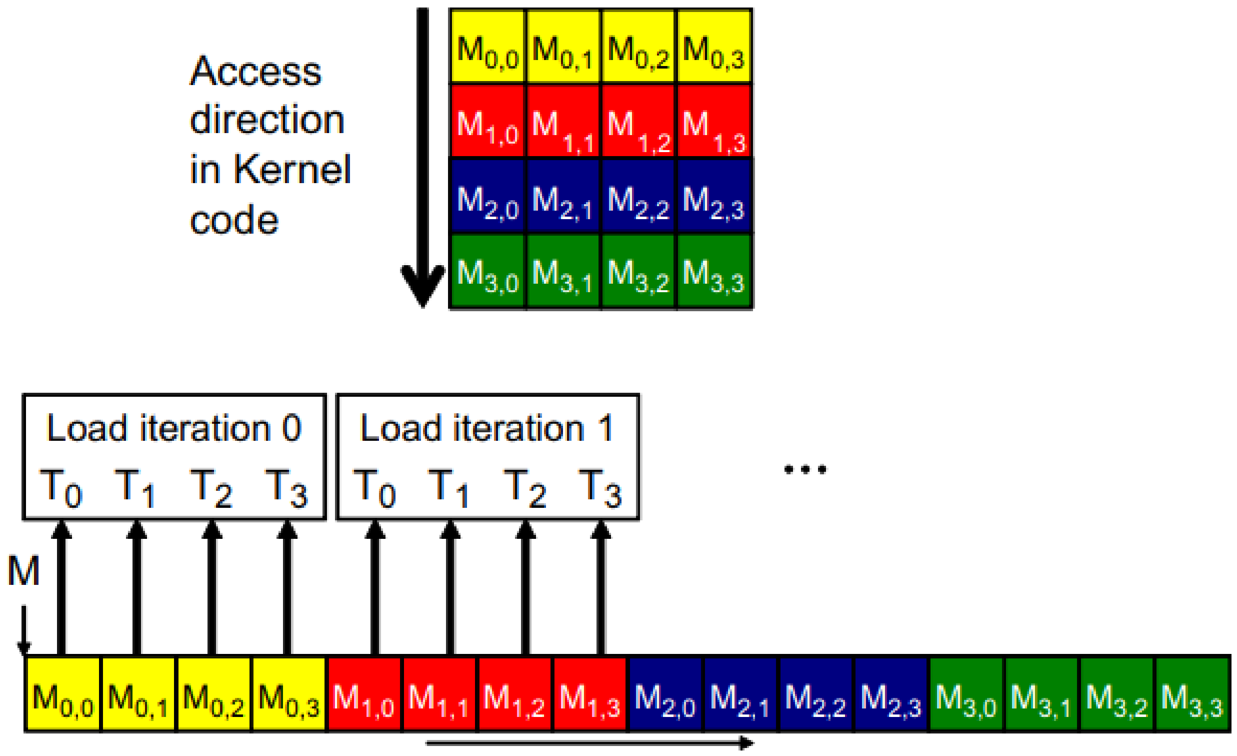
linearized order in increasing address

GLOBAL MEMORY BANDWIDTH

- Favorable versus unfavorable CUDA kernel 2D row-major array data access patterns for memory coalescing
  - During iteration 0, threads in a warp read element 0 of rows 0-31 During iteration 1, these same threads read element 1 of rows 0-31 None of the accesses will be coalesced
  - During iteration 0, threads in warp 0 read element 1 of columns 0-31. All these accesses will be coalesced



GLOBAL MEMORY BANDWIDTH



Access direction in Kernel code

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Load iteration 0: T_0 T_1 T_2 T_3

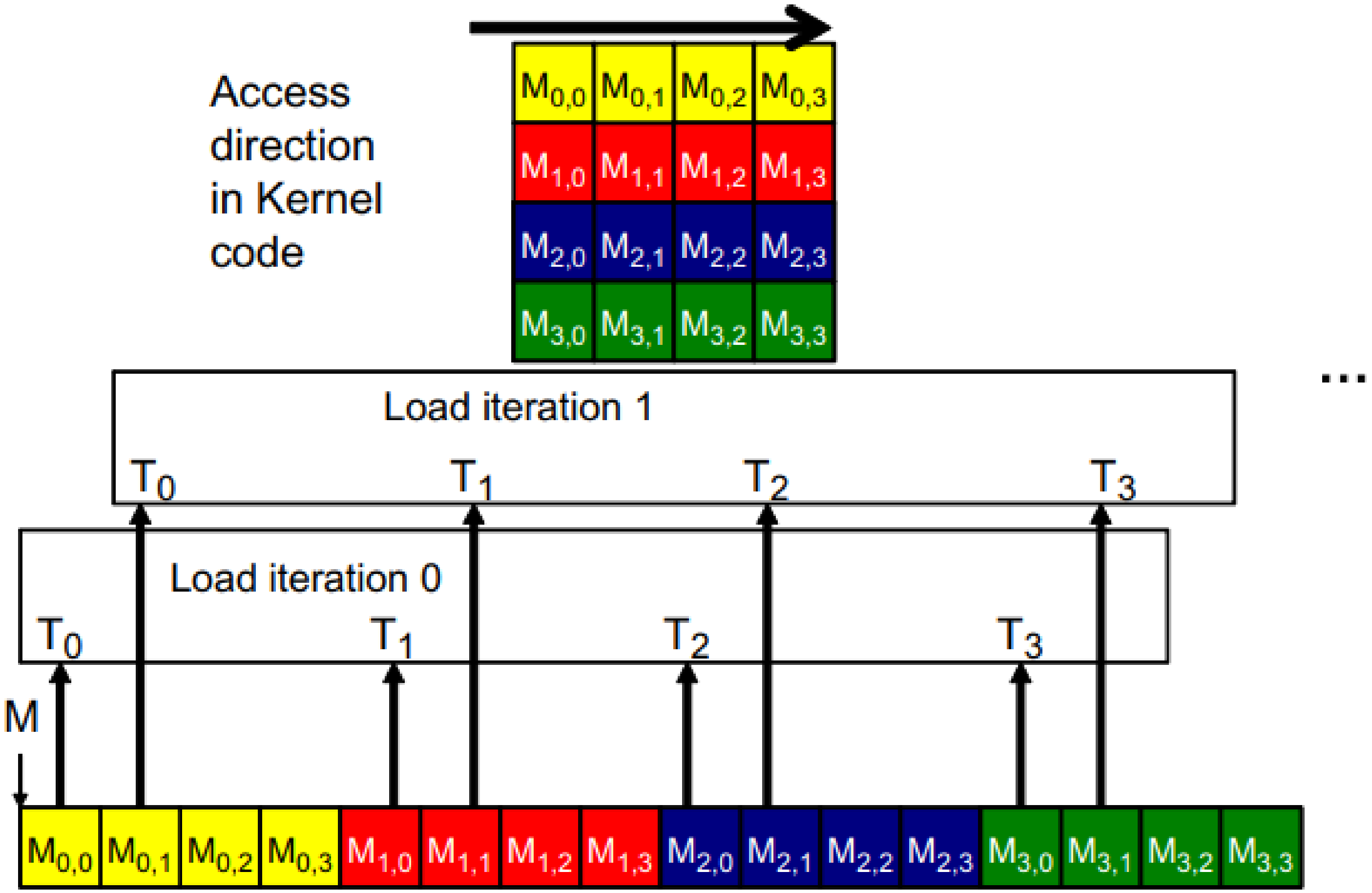
Load iteration 1: T_0 T_1 T_2 T_3

...

M

$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

GLOBAL MEMORY BANDWIDTH


Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Load iteration 1
T_0 T_1 T_2 T_3
…
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

GLOBAL MEMORY BANDWIDTH

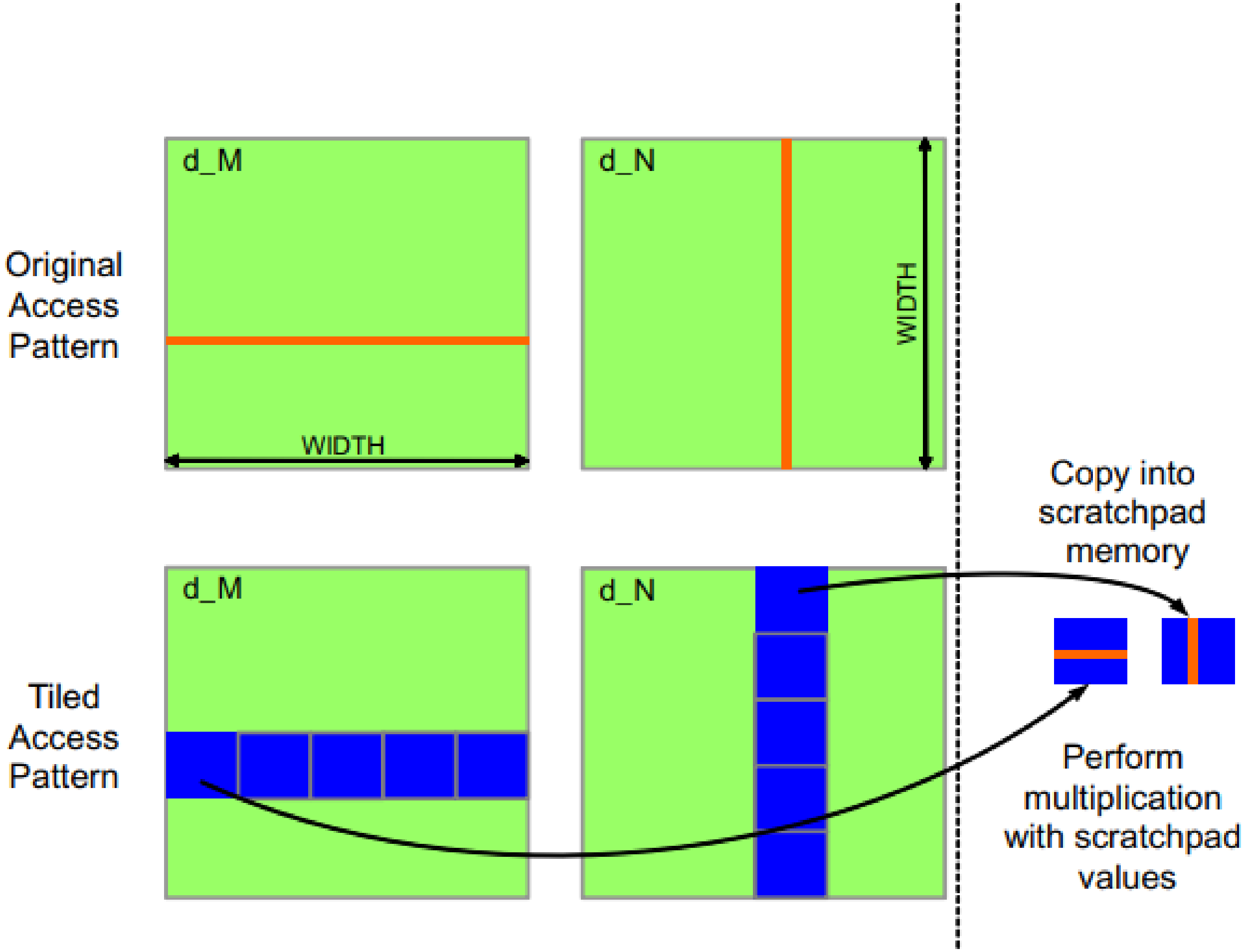
- A realistic matrix, there are typically hundreds or even thousands of elements in each dimension
- The elements accessed in each iteration by neighboring threads can be hundreds or even thousands of elements apart
- The hardware will determine that accesses to these elements are far away from each other and **cannot be coalesced**
- As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column

GLOBAL MEMORY BANDWIDTH

- If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the **shared memory to enable memory coalescing**
- Each thread reads a row from d_M (**Cannot be coalesced**)

- A **tiled algorithm** can be used to enable coalescing
- Threads of a block can first cooperatively load the tiles into the shared memory
- Care must be taken to ensure that these tiles are **loaded in a coalesced pattern**

GLOBAL MEMORY BANDWIDTH


Original Access Pattern
d_M
WIDTH
d_N
WIDTH
Tiled Access Pattern
d_M
d_N
Copy into scratchpad memory
Perform multiplication with scratchpad values

GLOBAL MEMORY BANDWIDTH

- Once the data is in shared memory, it can be accessed either on a **row basis or a column basis** with much less performance variation because the shared memories are implemented as intrinsically **high-speed, on-chip memory** that does not require coalescing to achieve a high data access rate
- The index calculation **d_M[row][m*TILE_SIZE + tx]** makes these threads access elements in the same row
- Elements in the same row are placed into consecutive locations of the global memory