Introduction to CUDA

Introduction

- The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism.
    - Because data parallelism plays such an important role in CUDA,

DATA PARALLELISM

- ==software applications that process a large amount of data== and thus incur ==long execution time.==
  - Model real-world, physical phenomena.
  - Images and video frames are snapshots model real-world, physical phenomena. Images and video frames are snapshots
- Physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps

" Independent evaluation is the basis of data parallelism in these applications."

Data parallelism

- Data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner.

  - " Note that the dot product operations for computing different matrix P elements can be simultaneously performed."

  - 1000 1000 matrix multiplication has 1,000,000 independent dot products

  - The data parallelism in real applications is not always as simple as that in our matrix multiplication
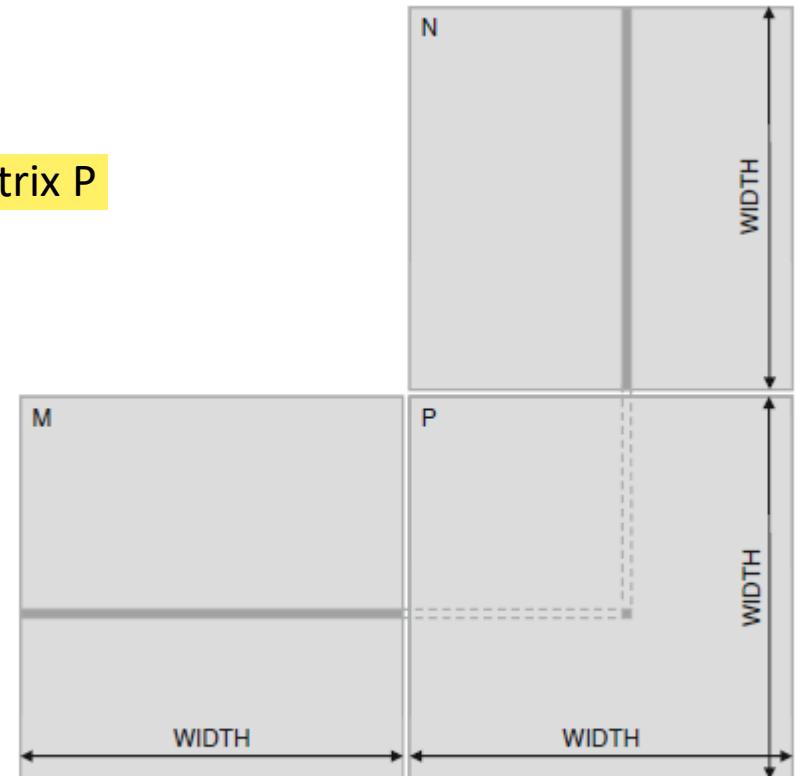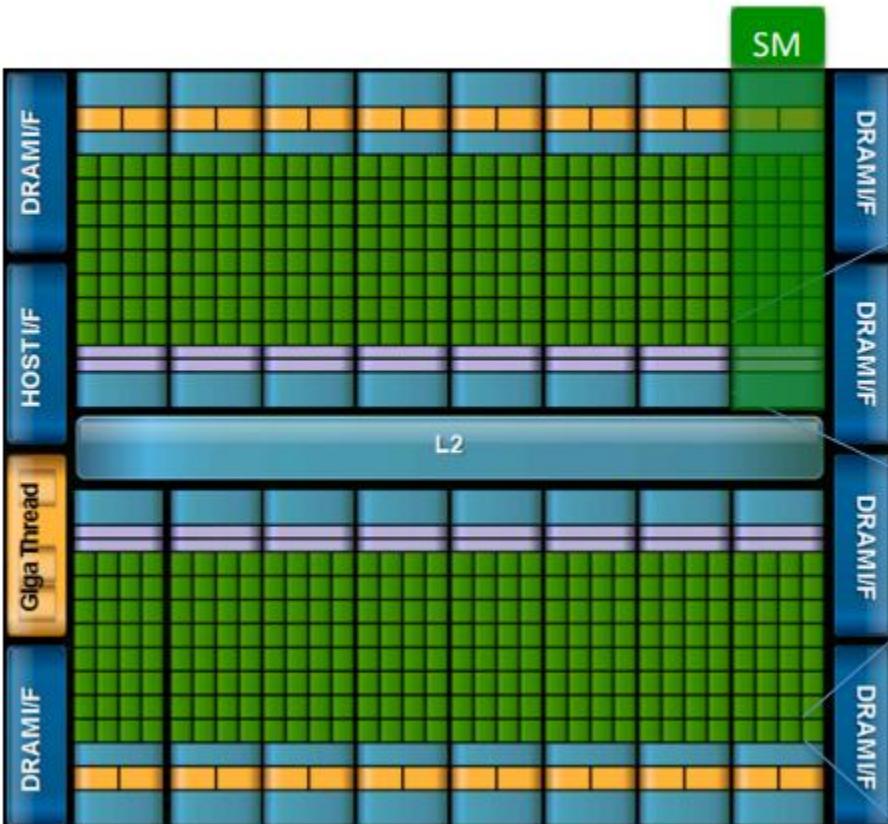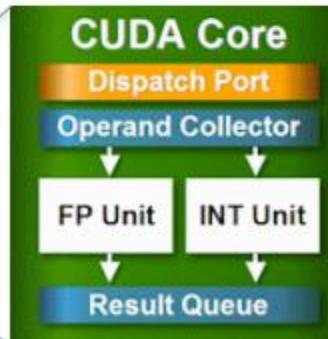
FIGURE 3.1

NVIDIA FERMI

512 CUDA cores (32/SM)
IEEE 754-2008 floating point (DP and SP)
6 GB GDDR5 DRAM (Global Memory)
ECC Memory support
Two DMA interface
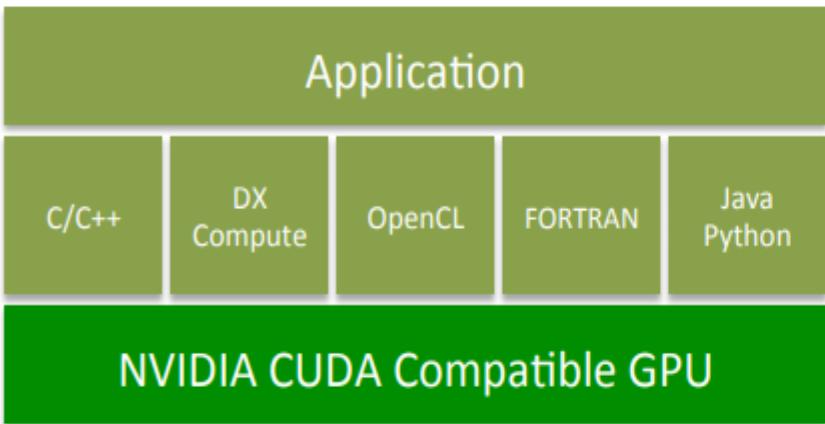
Reconfigurable L1 Cache and Shared Memory 48 KB / 16 KB

L2 Cache 768 KB
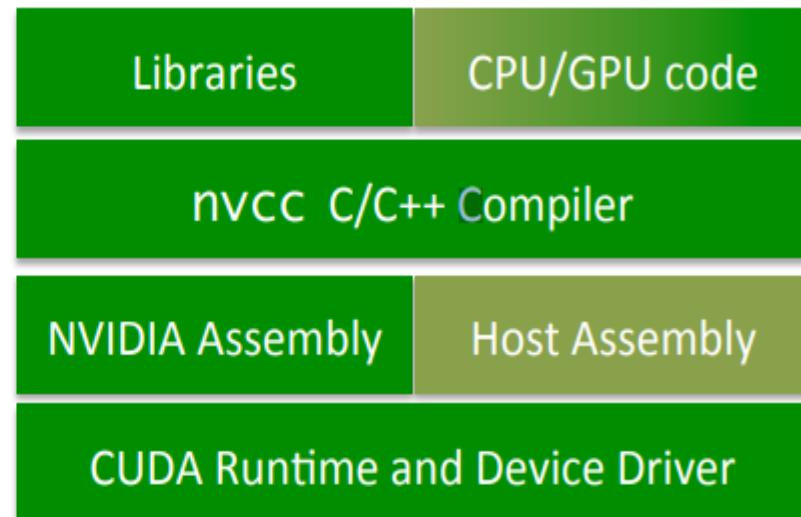
Load/Store address width 64 bits. Can calculate addresses of 16 threads per clock.
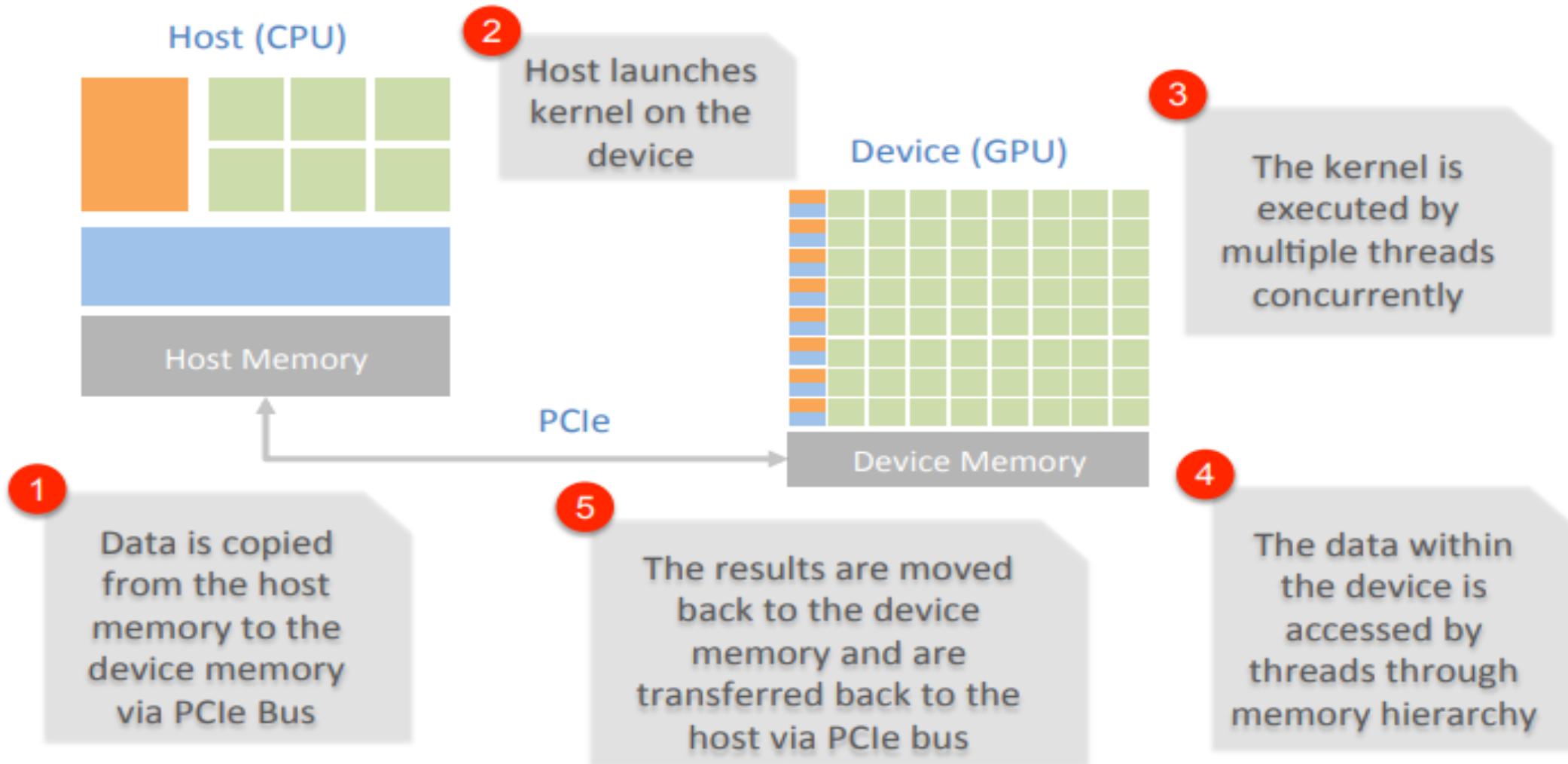
Dataflow

Host (CPU)

2 Host launches kernel on the device

Device (GPU)

3 The kernel is executed by multiple threads concurrently

Host Memory

PCIe

Device Memory

1 Data is copied from the host memory to the device memory via PCIe Bus

5 The results are moved back to the device memory and are transferred back to the host via PCIe bus

4 The data within the device is accessed by threads through memory hierarchy

Ref: https://iccs.lbl.gov/research/isaac/docs/IntroductiontoCUDAprogramming.pdf

CUDA PROGRAM STRUCTURE

- A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU.

- The phases that exhibit little or no data parallelism are implemented in host code.

- The phases that exhibit rich amount of data parallelism are implemented in the device code.

- A CUDA program is a unified source code encompassing both host and device code.

The NVIDIA C compiler (nvcc) separates the two during the compilation process.

1. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process.
2. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels
3. The device code is typically further compiled by the nvcc and executed on a GPU device

- The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism.
  - Ex: Entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix

- Number of threads used by the kernel is a function of the matrix dimension
- For a 1000 x 1000 matrix multiplication, the kernel that uses one thread to compute one P element would generate 1,000,000 threads when it is invoked.
- CUDA threads are of much lighter weight than the CPU threads
  - CUDA requires lower cycles to generate and schedule compared to CPU.

- All the threads that are generated by a kernel during an invocation are collectively called a **grid**

CPU serial code

GPU parallel kernel
KernelA<<< nBIK, nTid >>>(args);

Grid 0

CPU serial code

Grid 1

GPU parallel kernel
KernelA<<< nBIK, nTid >>>(args);
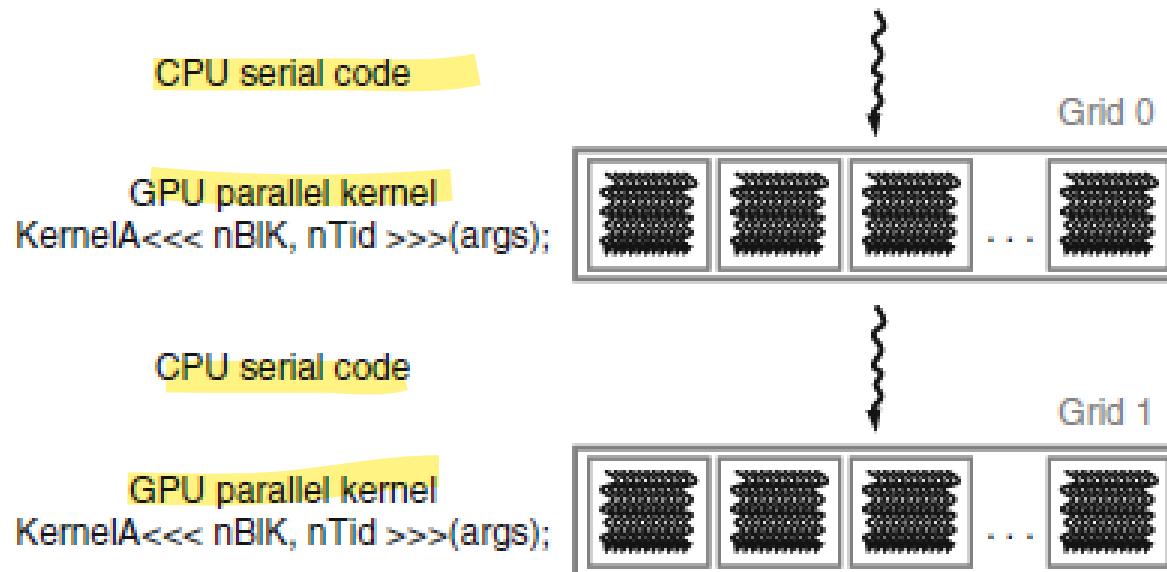
FIGURE 3.2

Execution of two grids of threads.
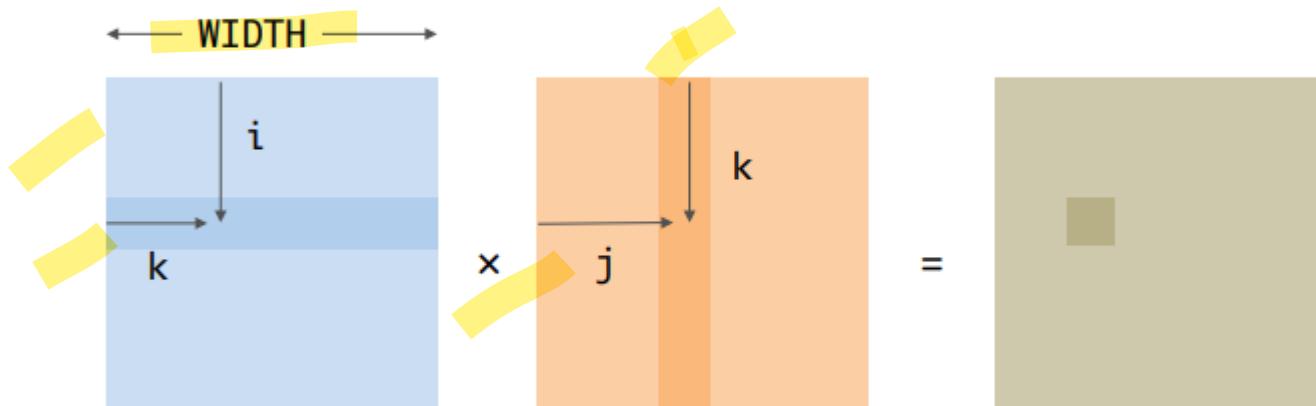
Execution of a CUDA program.

A MATRIX–MATRIX MULTIPLICATION EXAMPLE

We assume that the matrices are square in shape, and the dimension of each matrix is specified by the parameter Width
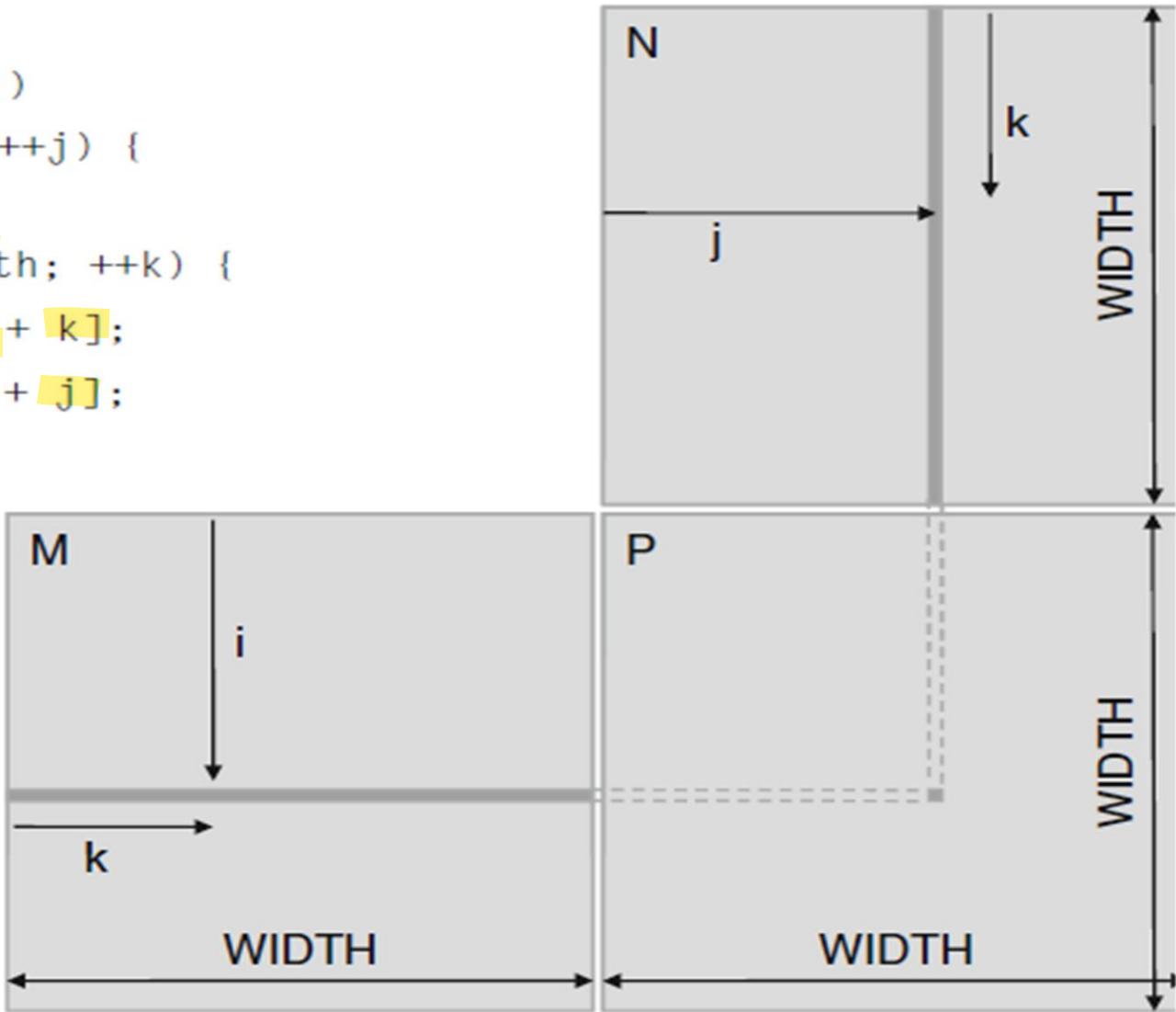
Example

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```
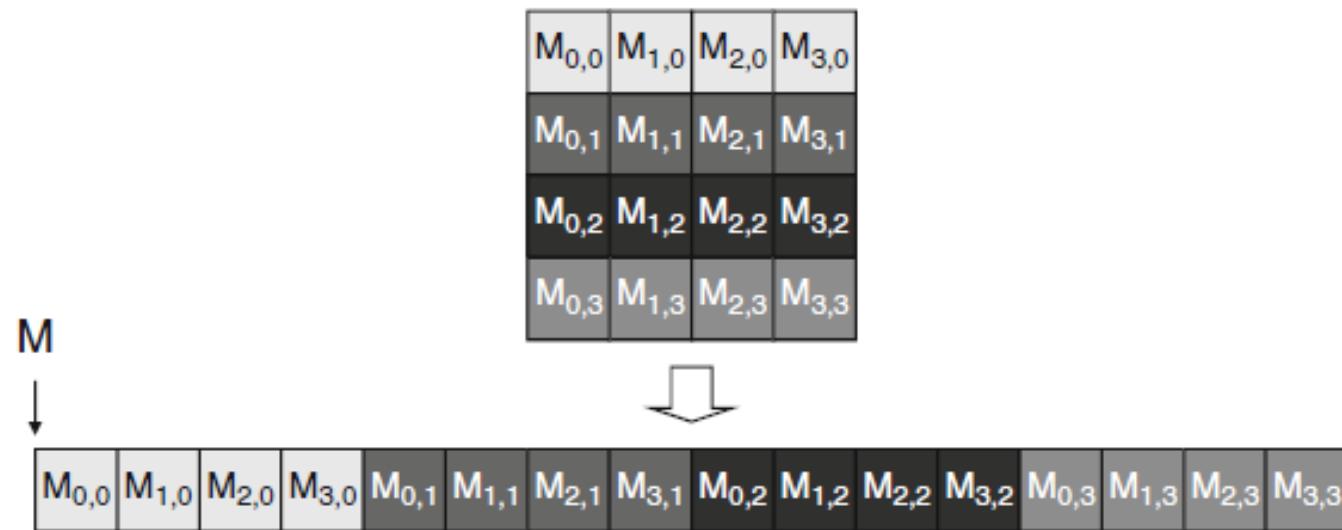
FIGURE 3.5

Placement of two-dimensional array elements into the linear address system memory.

DEVICE MEMORIES AND DATA TRANSFER

- In CUDA, the host and devices have separate memory spaces.
- This reflects the reality that devices are typically hardware cards that come with their own dynamic random access memory (DRAM).
- For example, the NVIDIA T10 processor comes with up to 4 GB (billion bytes, or gigabytes) of DRAM.
- In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory.

- 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
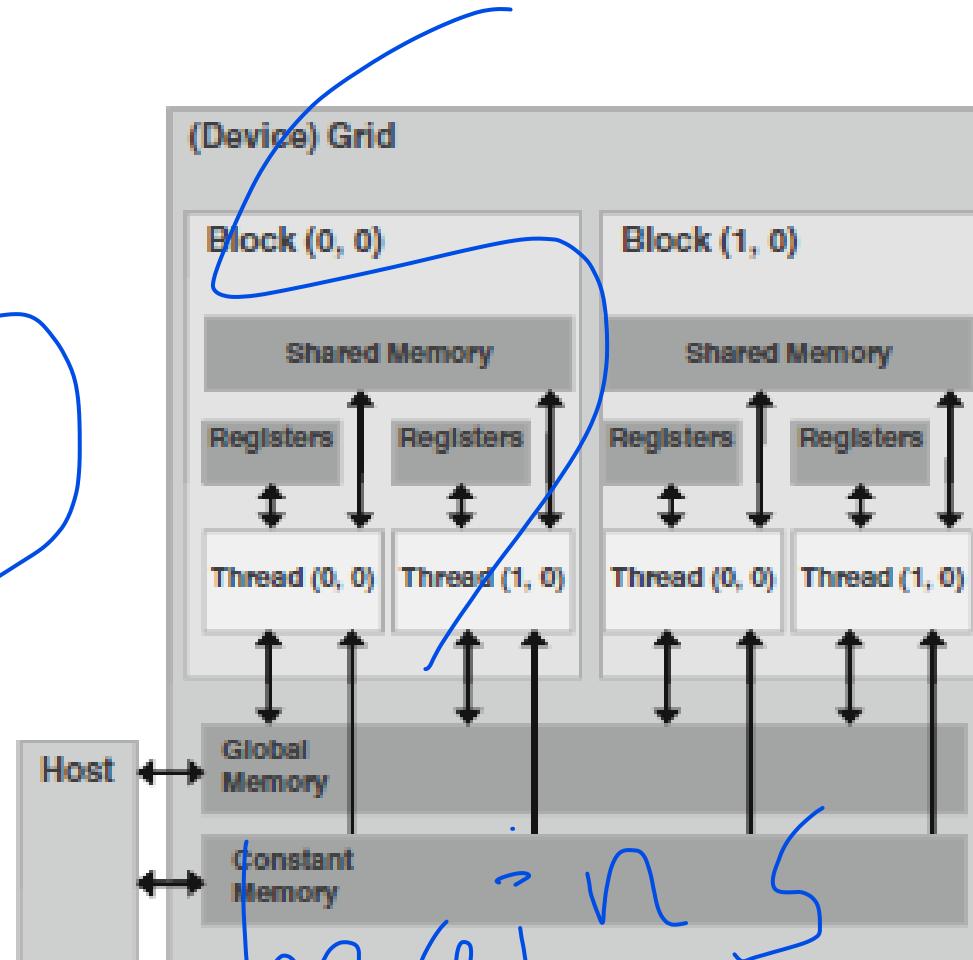
FIGURE 3.7

Overview of the CUDA device memory model.

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
   - Pointer to freed object
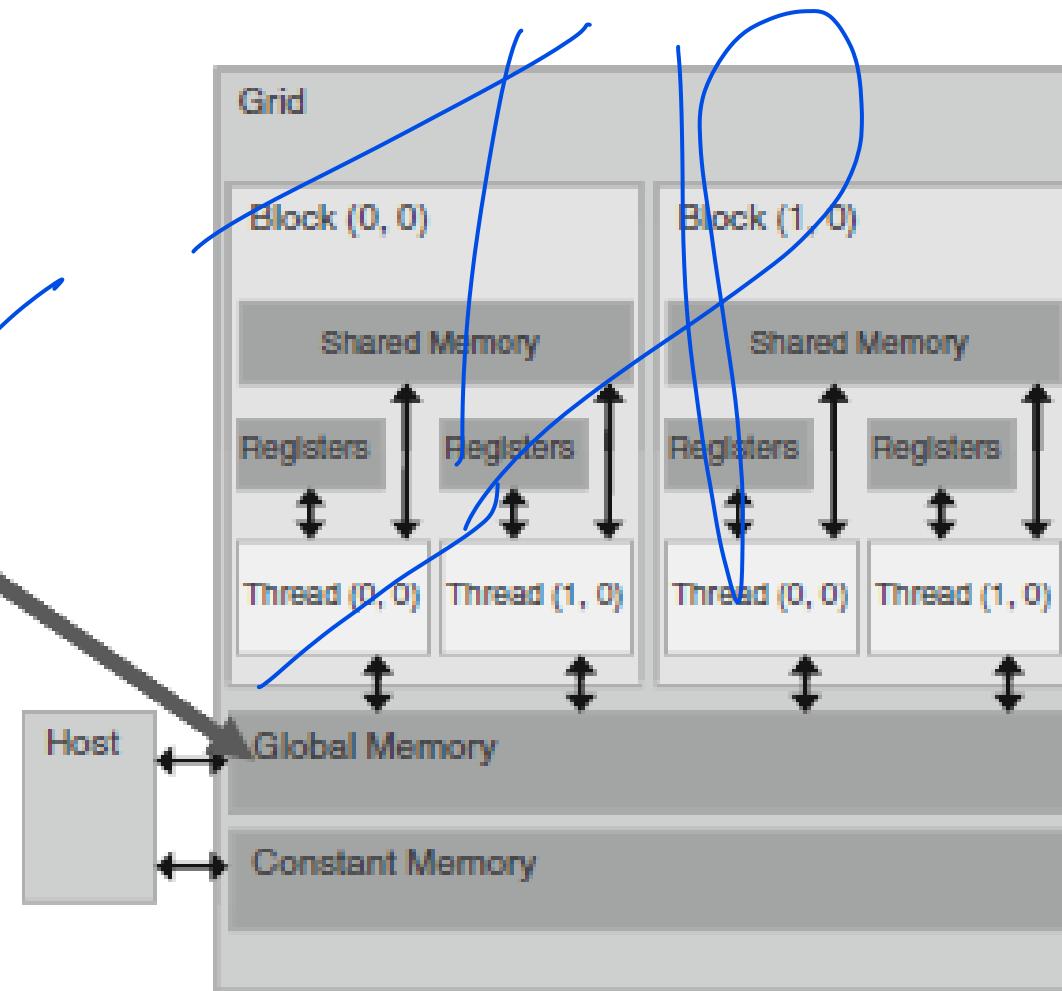
FIGURE 3.8

CUDA API functions for device global memory management.

- float *Md int size =Width * Width * sizeof(float);

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

- cudaFree(Md);

KERNEL FUNCTIONS AND THREADING

- In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Because all of these threads execute the same code, CUDA programming is an instance of the well-known single-program, multiple-data (SPMD)

- The syntax is ANSI C with some notable extensions. First, there is a CUDA-specific keyword "__global__" in front of the declaration of MatrixMulKernel(). This keyword indicates that the function is a kernel and that it can be called from a host functions to generate a grid of threads on a device.

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

	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 extensions to C functional declaration.

- A thread block is a batch
  of threads that can
  cooperate with each
  other by:

  — Synchronizing their
    execution
    - For hazard-free shared
      memory accesses

  — Efficiently sharing data
    through a low-latency
    shared memory

- Two threads from two
 different blocks cannot
  cooperate
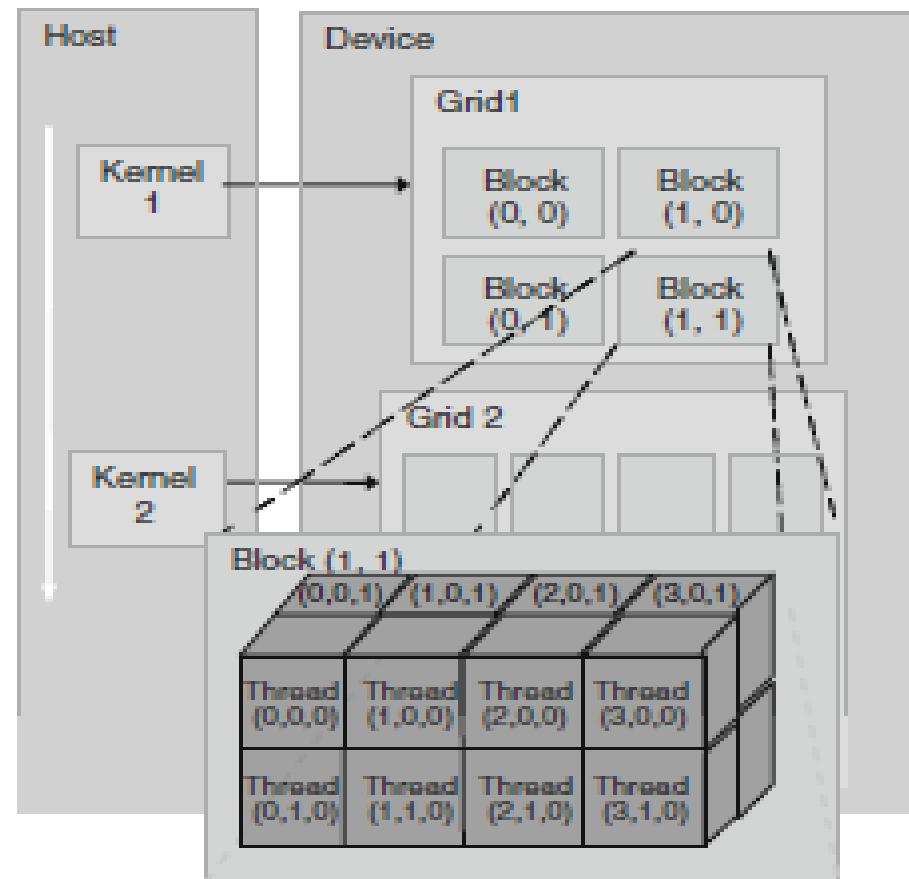
FIGURE 3.13

CUDA thread organization.

Unit-5

CUDA Threads

Contents

4.1 CUDA Thread Organization

 4.2 Using blockIdx and threadIdx

4.3 Synchronization and Transparent Scalability

4.4 Thread Assignment

4.5 Thread Scheduling and Latency Tolerance

CUDA THREAD ORGANIZATION

- Because all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process.

- These threads are organized into a two-level hierarchy using unique coordinates—blockIdx (for block index) and threadIdx (for thread index)—assigned to them by the CUDA runtime system.

- The blockIdx and threadIdx appear as built in, pre initialized variables that can be accessed within kernel functions.

- When a thread executes the kernel function, references to the blockIdx and threadIdx variables return the coordinates of the thread.
- Additional built-in variables, gridDim and blockDim, provide the dimension of the grid and the dimension of each block respectively.
- The black box of each thread block in Figure 4.1 shows a fragment of the kernel code.
- The code fragment uses the threadID = blockIdx.x * blockDim.x + threadIdx to identify the part of the input data to read from and the part of the output data structure to write to.
- Thread 3 of Block 0 has a threadID value of 0*M + 3 = 3.
- Thread 3 of Block 5 has a threadID value of 5*M + 3.

- Assume a grid has 128 blocks (N = 128) and each block has 32 threads (M = 32).

  In this example, access to blockDim in the kernel returns 32.
- There are a total of 128*32 = 4096 threads in the grid.
- Thread 3 of Block 0 has a threaded value of 0*32 + 3 =3.
- Thread 3 of Block 5 has a threaded value of 5*32 + 3 =163.
- Thread 15 of Block 102 has a threaded value of 3279.
- The reader should verify that every one of the 4096 threads has its own unique threaded value.
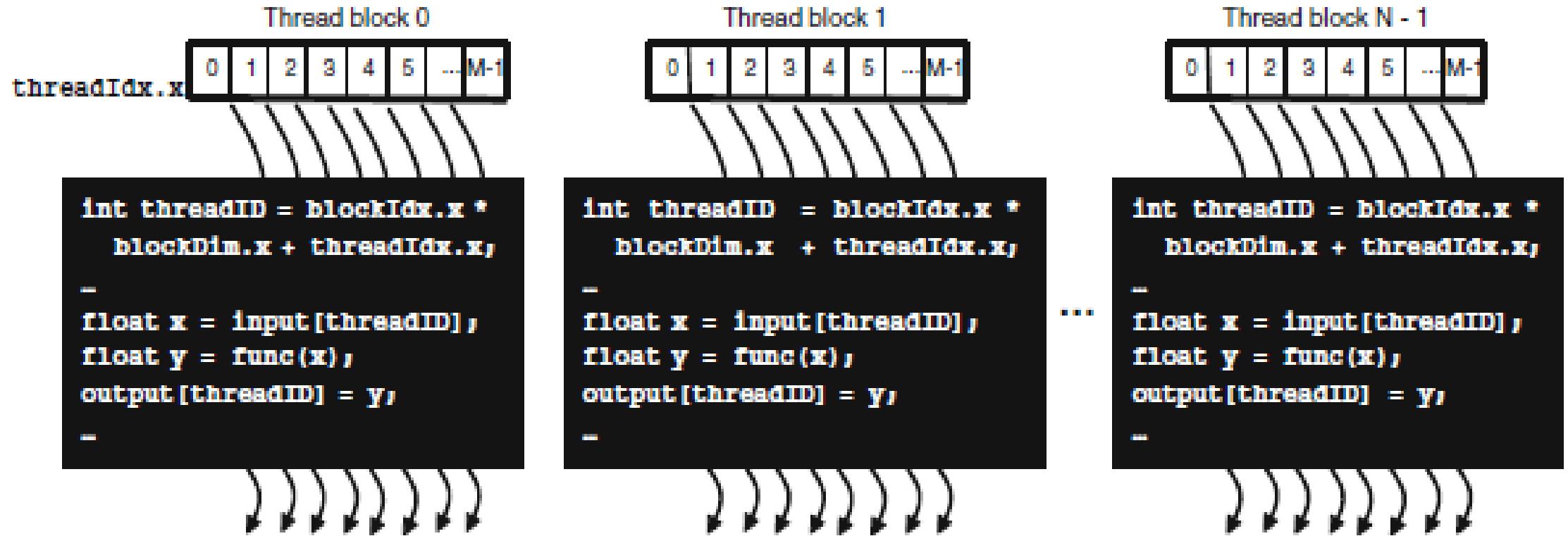
FIGURE 4.1

Overview of CUDA thread organization.

- In general, a grid is organized as a 2D array of blocks.
- Each block is organized into a 3D array of threads. The exact organization of a grid is determined by the execution configuration provided at kernel launch.
- The first parameter of the execution configuration specifies the dimensions of the grid in terms of number of blocks.
- The second specifies the dimensions of each block in terms of number of threads.
- Each such parameter is a dim3 type, which is essentially a C struct with three unsigned integer fields: x, y, and z. Because grids are 2D arrays of block dimensions, the third field of the grid dimension parameter is ignored; it should be set to 1 for clarity.

- dim3 dimGrid(128, 1, 1);
- dim3 dimBlock(32, 1, 1);
- KernelFunction<<<dimGrid, dimBlock>>>(. . .);

- KernelFunction<<<128, 32>>>(. . .);

- The values of gridDim.x and gridDim.y can range from 1 to 65,535.
- The values of gridDim.x and gridDim.y can be calculated based on other variables at kernel launch time.
- Once a kernel is launched, its dimensions cannot change.
- All threads in a block share the same blockIdx value.
- The blockIdx.x value ranges between 0 and gridDim.x 1, and the blockIdx.y value between 0 and gridDim.y 1
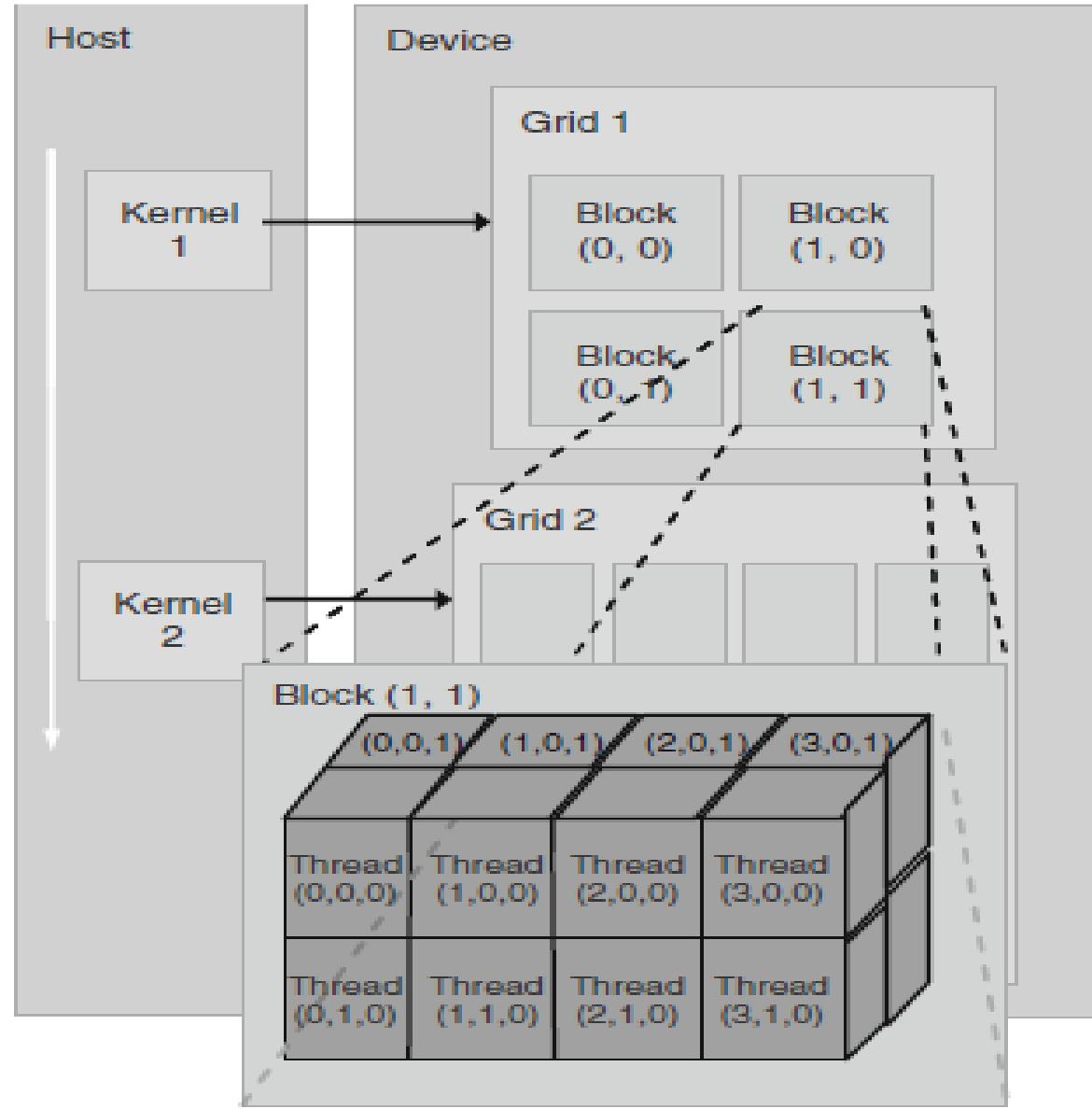
FIGURE 4.2

A multidimensional example of CUDA grid organization.

- Figure 4.2 shows a small 2D grid that is launched with the following
- host code:
- dim3 dimGrid(2, 2, 1);
- dim3 dimBlock(4, 2, 2);
- KernelFunction<<<dimGrid, dimBlock>>>(. . .);
- The grid consists of four blocks organized into a 22 array. Each block
- in Figure 4.2 is labeled with (blockIdx.x, blockIdx.y); for example,
- Block(1,0) has blockIdx.x ¼ 1 and blockIdx.y ¼ 0.

USING blockIdx AND threadIdx

- From the programmer's point of view, the main functionality of blockIdx and threadIdx variables is to provide threads with a means to distinguish among themselves when executing the same kernel.

- One common usage for threadIdx and blockIdx is to determine the area of data that a thread is to work on.

- This was illustrated by the simple matrix multiplication code in Figure 3.11, where the dot product loop uses threadIdx.x and threadIdx.y to identify the row of Md and column of Nd to work on.

- We will now cover more sophisticated usage of these variables.

- In order to accommodate larger matrices, we need to use multiple thread blocks.

- Figure 4.3 shows the basic idea of such an approach.

- Conceptually, we break Pd into square tiles. All the Pd elements of a tile are computed by a block of threads. By keeping the dimensions of these Pd tiles small, we keep the total number of threads in each block under 512, the maximal allowable block size.

- In Figure 4.3, for simplicity, we abbreviate threadIdx.x and threadIdx.y as tx and ty. Similarly, we abbreviate blockIdx.x and blockIdx.y as bx and by.

- Each thread still calculates one Pd element.
- The difference is that it must use its blockIdx values to identify the tile that contains its element before it uses its threadIdx values to identify its element inside the tile.
- That is, each thread now uses both threadIdx and blockIdx to identify the Pd element to work on.
- This is portrayed in Figure 4.3, where the bx, by, tx, and ty values of threads calculating the Pd elements are marked in both x and y dimensions.
- All threads calculating the Pd elements within a tile have the same blockIdx values.

- Assume that the dimensions of a block are square and are specified by the variable TILE_WIDTH.
- Each dimension of Pd is now divided into sections of TILE_WIDTH elements each, as shown on the left and top edges of Figure 4.3.
- Each block handles such a section. Thus, a thread can find thex index of its Pd element as (bx*TILE_WIDTH + tx) and the y index as (by*TILE_WIDTH + ty).
- That is, thread (tx, ty) in block (bx, by) is to use row (by*TILE_WIDTH + ty) of Md and column (bx*TILE_WIDTH + tx) of Nd to calculate the Pd element at column (bx*TILE_WIDTH + tx) and row by*TILE_WIDTH + ty).
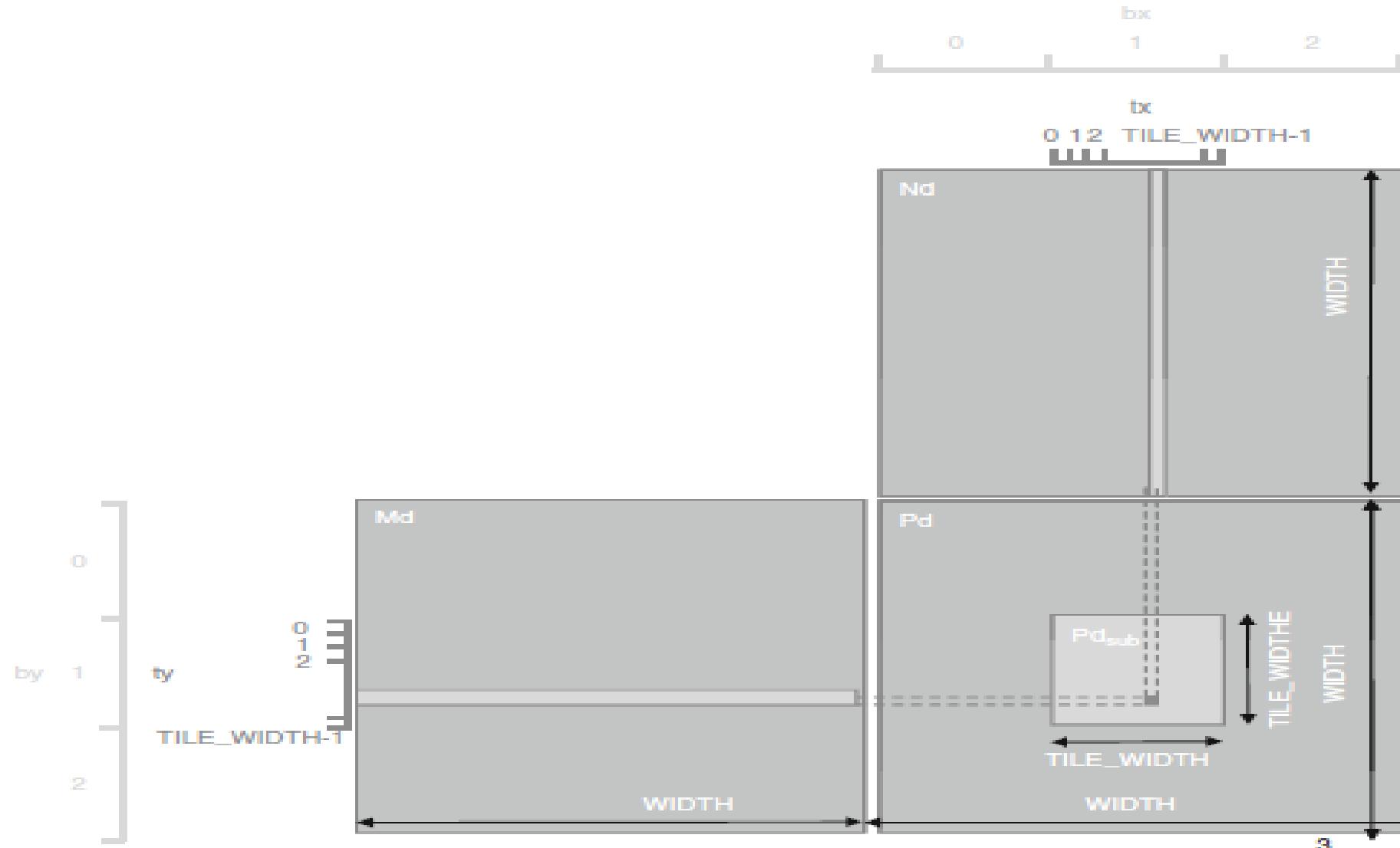
FIGURE 4.3

Matrix multiplication using multiple blocks by tiling **Pd**.
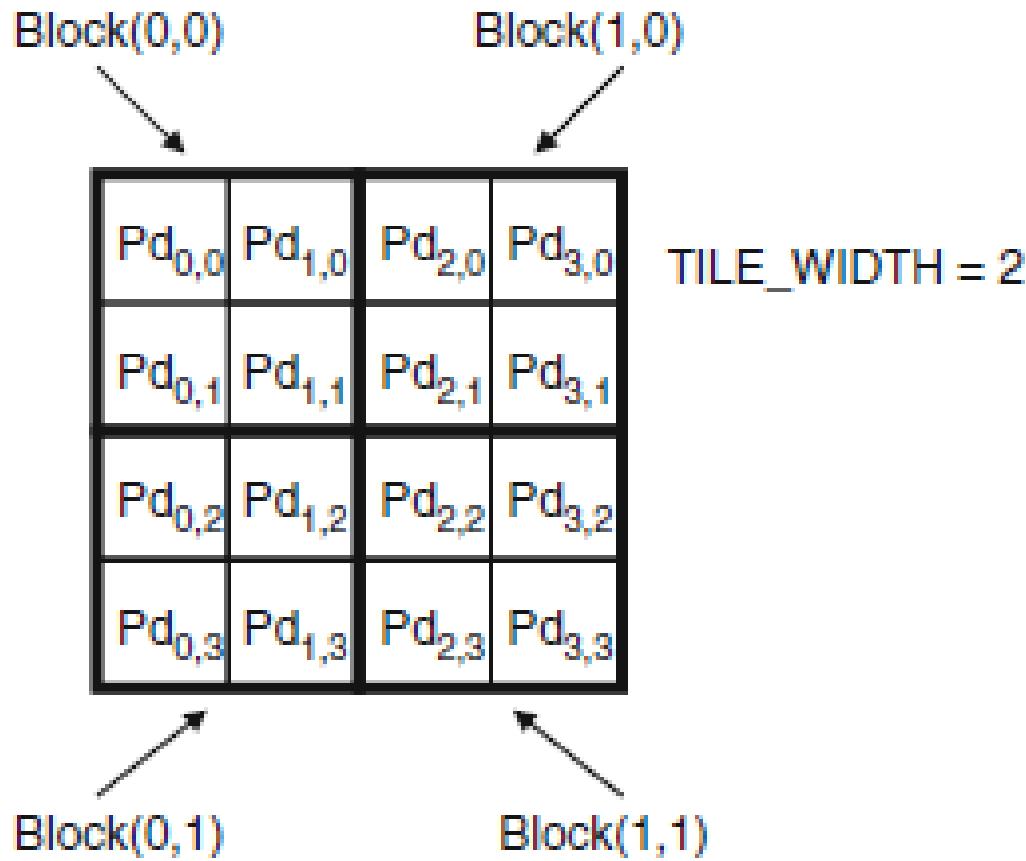
FIGURE 4.4

A simplified example of using multiple blocks to calculate **Pd**.

- Once we have identified the indices for the Pd element to be calculated by
- a thread, we also have identified the row (y) index of Md and the column (x) index of Nd for input values. As shown in Figure 4.3, the row index of Md used by thread (tx, ty) of block (bx, by) is (by*TILE_WIDTH þ ty).
- The column index of Nd used by the same thread is (bx*TILE_WIDTH þ tx).
- We are now ready to revise the kernel of Figure 3.11 into a version that uses multiple blocks to calculate Pd.

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(), the thread that executes the function call 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 moves on to the next phase.

- Barrier synchronization is a ==simple and popular method of coordinating parallel activities.==

- In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons; for example, assume that four friends go to a shopping mall in a car.

- They can all go to different stores to buy their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit the stores to shop for their clothes. However, barrier synchronization is needed before they leave the mall.

- They have to wait until all four friends have returned to the car before they can leave.

- Without barrier synchronization, one or more persons could be left at the mall when the car leaves, which could seriously damage their friendship!

- In CUDA, a __syncthreads() statement must be executed by all threads in a block.
- When a __syncthreads() statement is placed in an if statement, either all threads in a block execute the path that includes the __syncthreads() or none of them does.
- For an if–then–else statement, if each path has a __syncthreads() statement, then either all threads in a block execute the __syncthreads() on the then path or all of them execute the else path.
- The two __syncthreads() are different barrier synchronization points.
- If a thread in a block executes the then path and another executes the else path, then they would be waiting at different barrier synchronization points.
- They would end up waiting for each other forever.
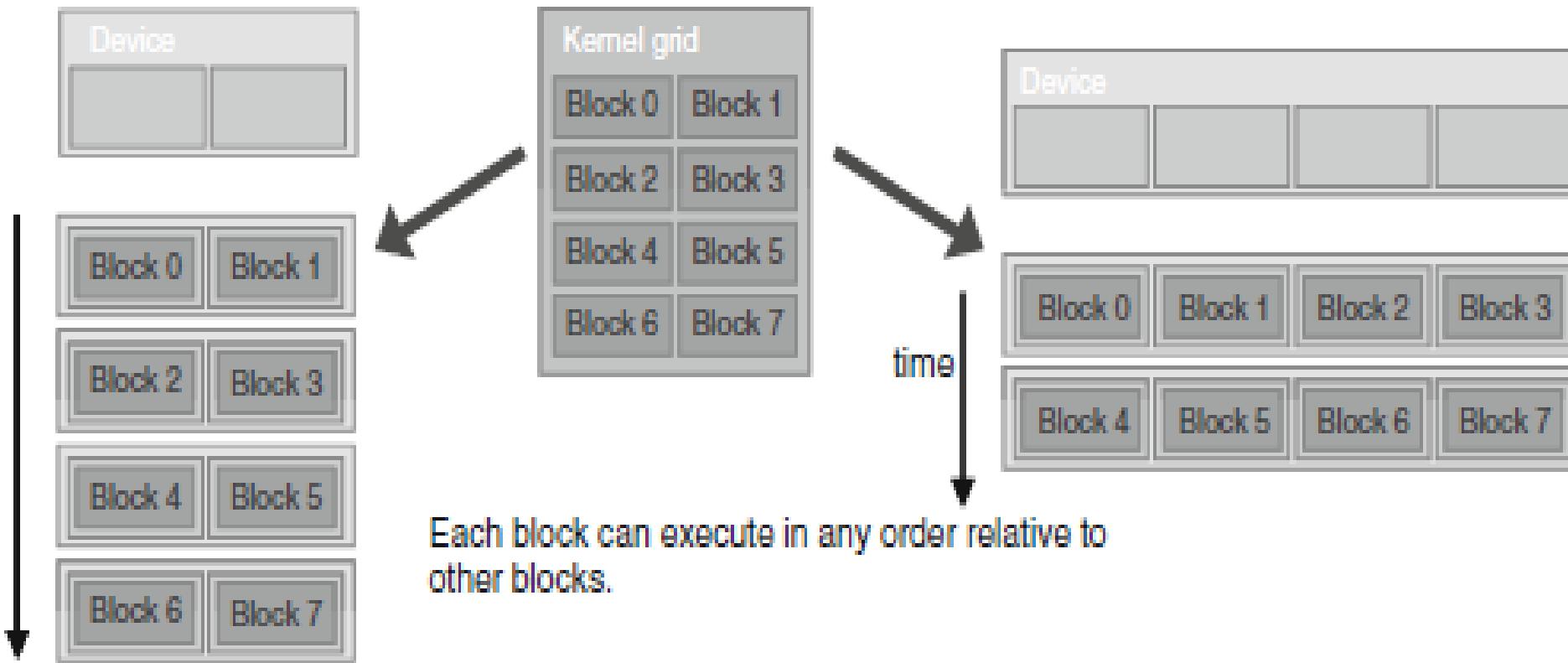
FIGURE 4.8

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

THREAD ASSIGNMENT

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. These 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);

- for example, the NVIDIA GT200 implementation has 30 streaming multiprocessors, 2 of which are shown in Figure 4.9. Up to 8 blocks can be assigned to each SM in the GT200 design as long as there are enough resources to satisfy the needs of all of the blocks.

- In situations with an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit.

- With 30 SMs in the GT200 processor, up to 240 blocks can be simultaneously assigned to them. Most grids contain many more than 240 blocks. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.
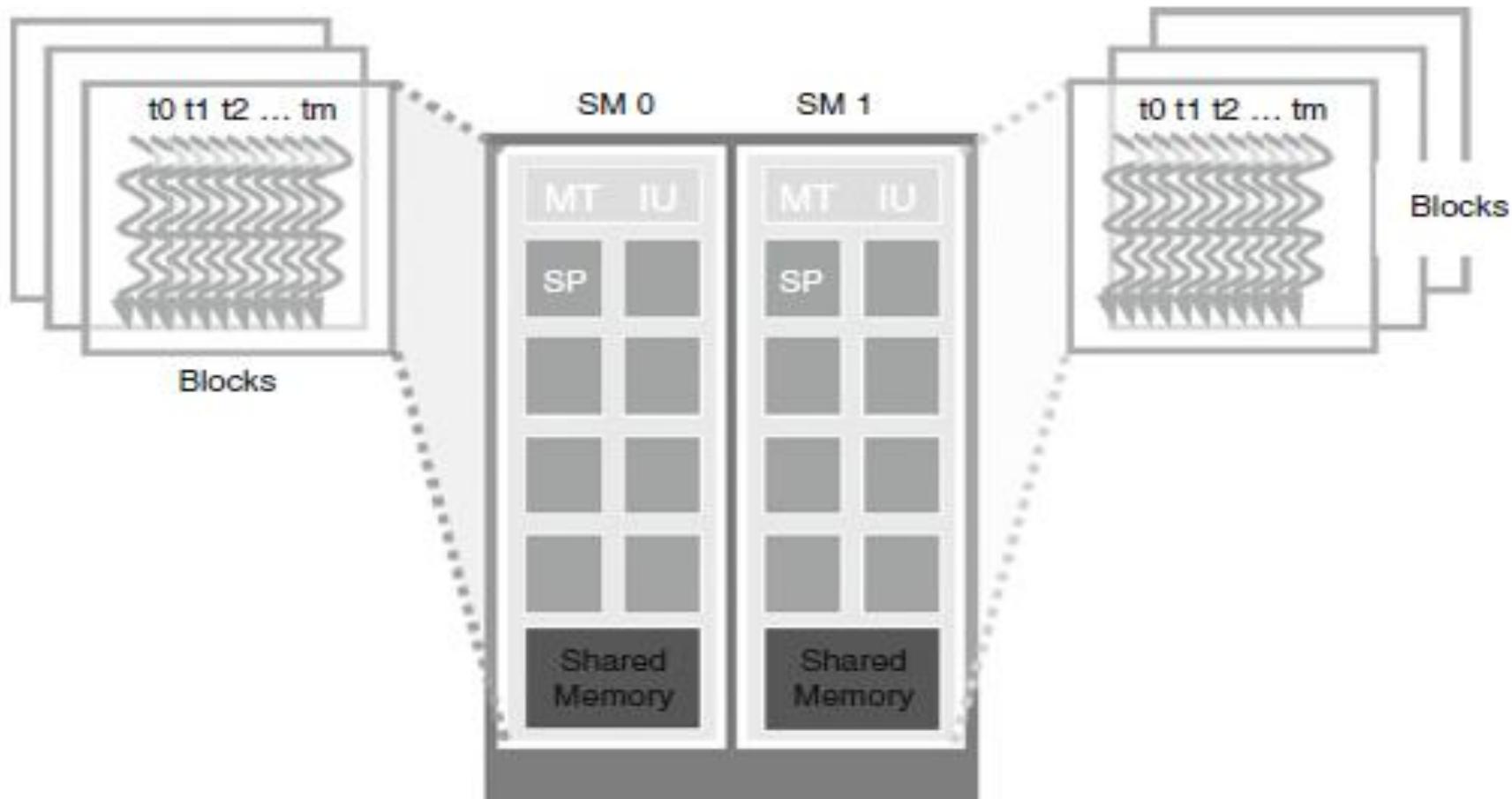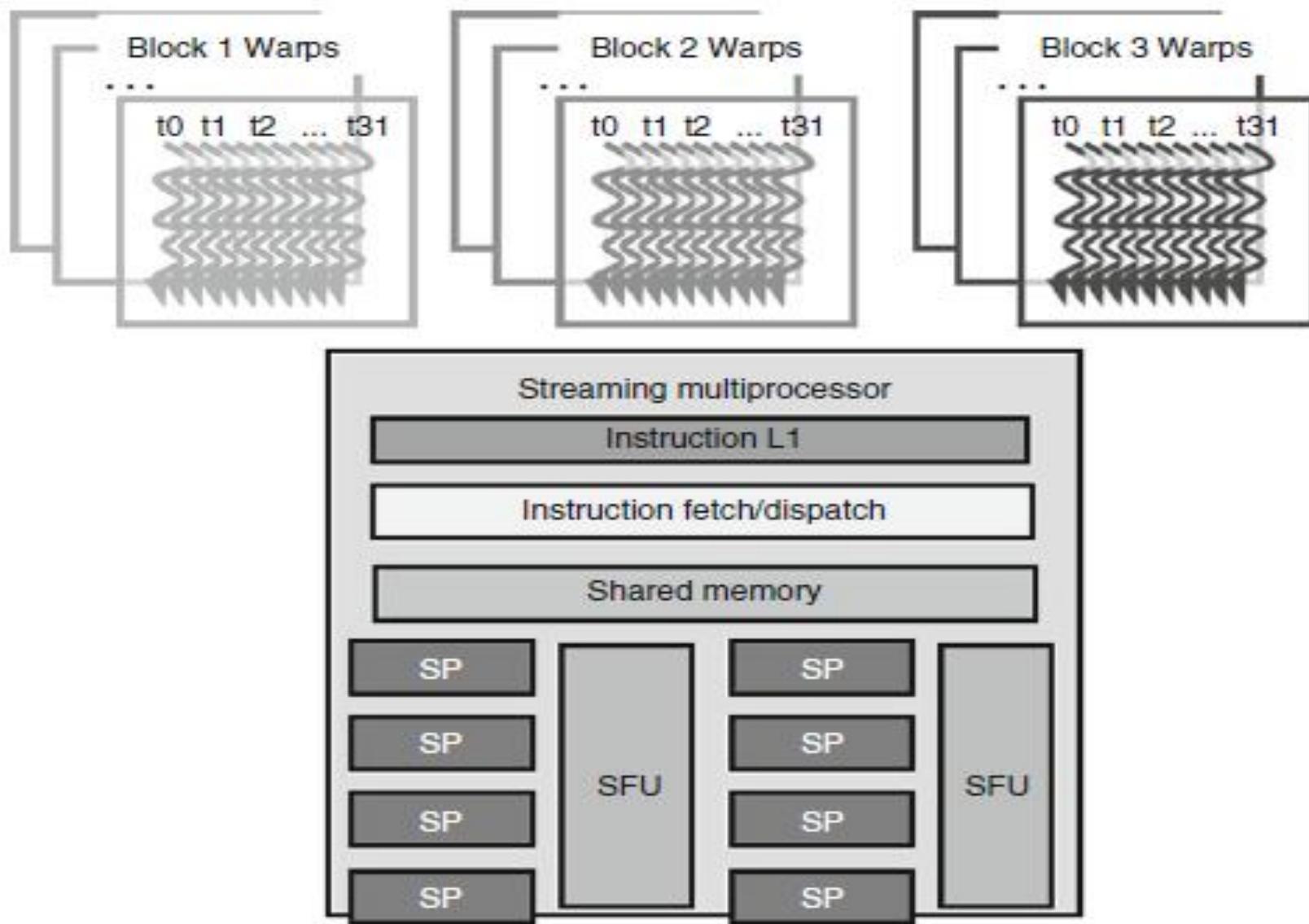
FIGURE 4.9

Thread block assignment to streaming multiprocessors (SMs).

THREAD SCHEDULING AND LATENCY TOLERANCE

- In the GT200 implementation, once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called warps.

- The size of warps is implementation specific.

- In fact, warps are not part of the CUDA specification; however, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices.

- The warp is the unit of thread scheduling in SMs

Block 1 Warps

t0 t1 t2 ... t31

Block 2 Warps

t0 t1 t2 ... t31

Block 3 Warps

t0 t1 t2 ... t31

Streaming multiprocessor

Instruction L1

Instruction fetch/dispatch

Shared memory

SP

SP

SP

SP

SFU

SP

SP

SP

SP

SFU

Chapter-5

CUDA Memories

CHAPTER CONTENTS

- 5.1 Importance of Memory Access Efficiency

- 5.2 CUDA Device Memory Types

- 5.3 A Strategy for Reducing Global Memory Traffic

- 5.4 Memory as a Limiting Factor to Parallelism

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- We can illustrate the effect of memory access efficiency by calculating the
- expected performance level of the matrix multiplication kernel code shown
- in Figure 4.6, replicated here in Figure 5.1.
- The most important part of the kernel in terms of execution time is the for loop that performs dot product calculations.
- In every iteration of this loop, two global memory accesses are performed for ==one floating-point multiplication== and ==one floating-point addition.==
- Thus, the ratio of floating-point calculation to the global memory access operation is 1 to 1, or 1.0. We will refer to this ratio 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.

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

FIGURE 5.1

Matrix multiplication kernel using multiple blocks (see Figure 4.6).

CUDA DEVICE MEMORY TYPES

- CUDA supports several types of memory that can be used by programmers to achieve high CGMA ratios and thus high execution speeds in their kernels.

- At the bottom of the figure, we see global memory and constant memory. These types of memory can be written (W) and read (R) by the host by calling application programming interface (API) functions.

- The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

- Registers and shared memory in Figure 5.2 are 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.
- A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.
- 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.

- 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
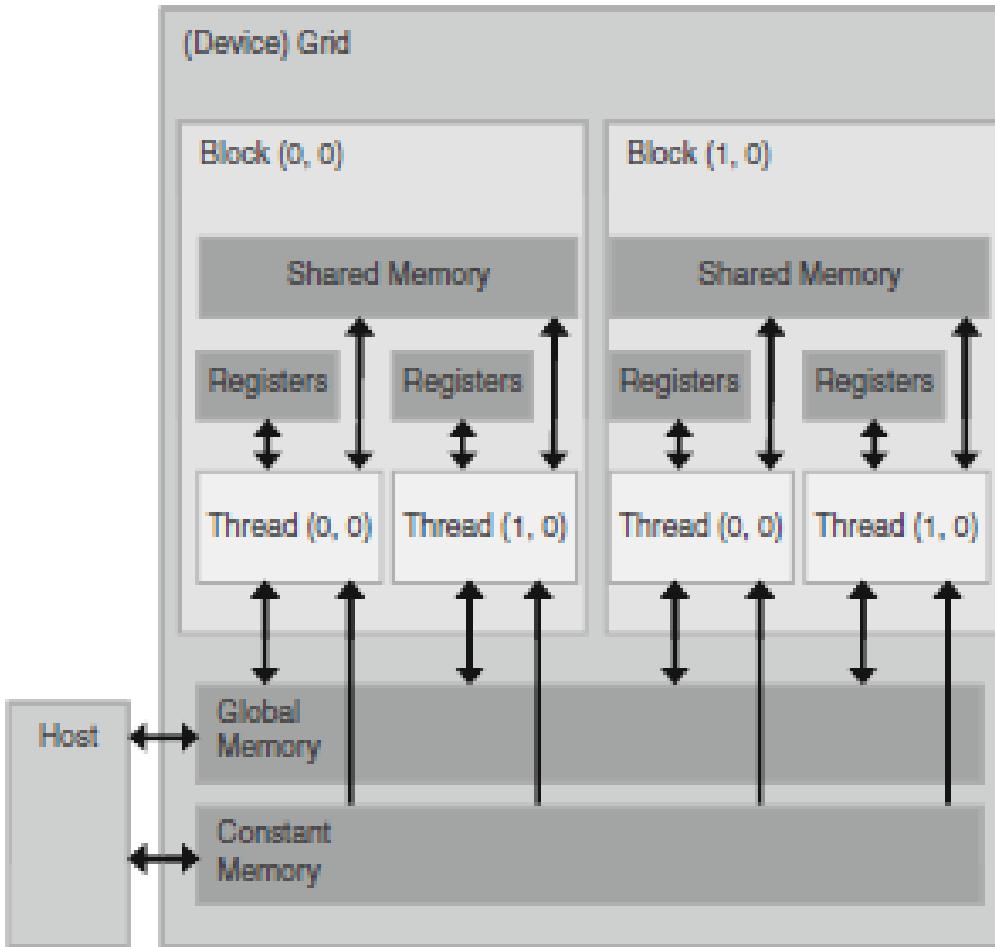
FIGURE 5.2

Overview of the CUDA device memory model.

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

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Global memory is large but slow, whereas the shared memory is small but fast.

- A common strategy is to partition the data into subsets called tiles such that each tile fits into the shared memory.

- The term tile draws on the analogy that a large wall (i.e., the global memory data) can often be covered by tiles (i.e., subsets that each can fit into the shared memory).

- An important criterion is that the kernel computations on these tiles can be done independently of each other.

- Note that not all data structures can be partitioned into tiles given an arbitrary kernel function

- The concept of tiling can be illustrated with the matrix multiplication example.
- Figure 5.3 shows a small example of matrix multiplication using multiple blocks.
- It corresponds to the kernel function in Figure 5.1. This example assumes that we use four 2 x2 blocks to compute the Pd matrix.
- Figure 5.3 highlights the computation done by the four threads of block (0, 0).
- These four threads compute Pd0,0, Pd1,0, Pd0,1, and Pd1,1.
- The accesses to the Md and Nd elements by thread (0, 0) and thread (1, 0) of block (0, 0) are highlighted with black arrows.
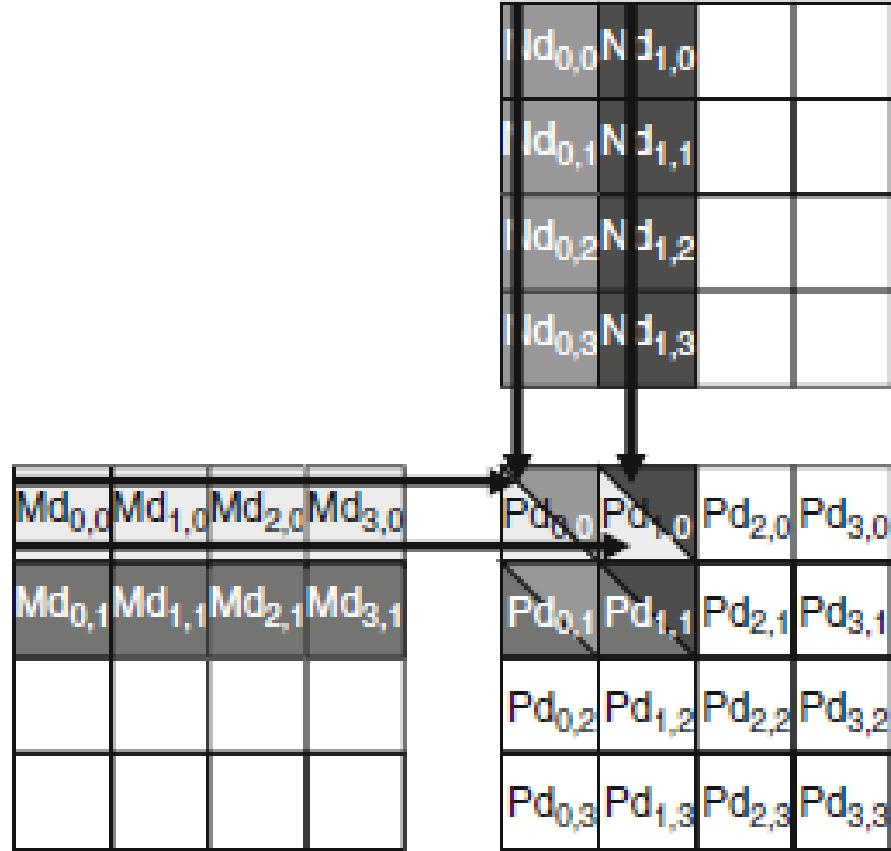
FIGURE 5.3

A small example of matrix multiplication using multiple blocks.

$Pd_{0,0}$ Thread(0,0)	$Pd_{1,0}$ Thread(1,0)	$Pd_{0,1}$ Thread(0,1)	$Pd_{1,1}$ Thread(1,1)
$Md_{0,0} * Nd_{0,0}$	$Md_{0,0} * Nd_{1,0}$	$Md_{0,1} * Nd_{0,0}$	$Md_{0,1} * Nd_{1,0}$
$Md_{1,0} * Nd_{0,1}$	$Md_{1,0} * Nd_{1,1}$	$Md_{1,1} * Nd_{0,1}$	$Md_{1,1} * Nd_{1,1}$
$Md_{2,0} * Nd_{0,2}$	$Md_{2,0} * Nd_{1,2}$	$Md_{2,1} * Nd_{0,2}$	$Md_{2,1} * Nd_{1,2}$
$Md_{3,0} * Nd_{0,3}$	$Md_{3,0} * Nd_{1,3}$	$Md_{3,1} * Nd_{0,3}$	$Md_{3,1} * Nd_{1,3}$

Access order

FIGURE 5.4

Global memory accesses performed by threads in block(0,0).

Pd$_{0,0}$ Thread(0,0)	Pd$_{1,0}$ Thread(1,0)	Pd$_{0,1}$ Thread(0,1)	Pd$_{1,1}$ Thread(1,1)
Md$_{0,0}$ * Nd$_{0,0}$	Md$_{0,0}$ * Nd$_{1,0}$	Md$_{0,1}$ * Nd$_{0,0}$	Md$_{0,1}$ * Nd$_{1,0}$
Md$_{1,0}$ * Nd$_{0,1}$	Md$_{1,0}$ * Nd$_{1,1}$	Md$_{1,1}$ * Nd$_{0,1}$	Md$_{1,1}$ * Nd$_{1,1}$
Md$_{2,0}$ * Nd$_{0,2}$	Md$_{2,0}$ * Nd$_{1,2}$	Md$_{2,1}$ * Nd$_{0,2}$	Md$_{2,1}$ * Nd$_{1,2}$
Md$_{3,0}$ * Nd$_{0,3}$	Md$_{3,0}$ * Nd$_{1,3}$	Md$_{3,1}$ * Nd$_{0,3}$	Md$_{3,1}$ * Nd$_{1,3}$

Access order

FIGURE 5.4

Global memory accesses performed by threads in block(0,0).

- Figure 5.4 shows the global memory accesses done by all threads in block(0,0).
- The threads are listed in the horizontal direction, with the time of access increasing downward in the vertical direction.
- Note that each thread accesses four elements of Md and four elements of Nd during its execution.
- Among the four threads highlighted, there is a significant overlap in terms of the Md and Nd elements they access;
- for example, thread(0,0) and thread(1,0) both access $Md_{1,0}$ as well as the rest of row 0 of Md.
- Similarly, thread1,0 and thread1,1 both access $Nd_{1,0}$ as well as the rest of column 1 of Nd.

- In Figure 5.5, we divide Md and Nd into 22 tiles, as delineated by the thick lines.

- 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 Md and a tile of Nd into the shared memory.

- This is done by having every thread in a block to load one Md element and one Nd element into the shared memory, as illustrated in Figure 5.6. Each row of Figure 5.6 shows the execution activities of a thread.

- We only need to show the activities of threads in block(0,0); the other blocks all have the same behavior.

- The shared memory array for the Md elements is called Mds. The shared memory array for the Nd elements is called Nds.

- At the beginning of Phase 1, the four threads of block(0,0) collaboratively load a tile of Md into shared memory; thread(0,0) loads $Md_{0,0}$ into $Mds_{0,0}$, thread(1,0) loads $Md_{1,0}$ into $Mds_{1,0}$, thread(0,1) loads $Md_{0,1}$ into $Mds_{0,1}$, and thread(1,1) loads $Md_{1,1}$ into $Mds_{1,1}$. Look at thesecond column of Figu re 5.6. A tile of Nd is also loaded in a similar manner, as shown in the third column of Figure 5.6.

FIGURE 5.5

Tiling **Md** and **Nd** to utilize shared memory.

	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ $\downarrow$ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ $\downarrow$ $Nds_{0,0}$	$PValue_{0,0}\ +=$ $Mds_{0,0}{}^*Nds_{0,0}\ +$ $Mds_{1,0}{}^*Nds_{0,1}$	$\mathbf{Md_{2,0}}$ $\downarrow$ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ $\downarrow$ $Nds_{0,0}$	$PValue_{0,0}\ +=$ $Mds_{0,0}{}^*Nds_{0,0}\ +$ $Mds_{1,0}{}^*Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ $\downarrow$ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ $\downarrow$ $Nds_{1,0}$	$PValue_{1,0}\ +=$ $Mds_{0,0}{}^*Nds_{1,0}\ +$ $Mds_{1,0}{}^*Nds_{1,1}$	$\mathbf{Md_{3,0}}$ $\downarrow$ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ $\downarrow$ $Nds_{1,0}$	$PValue_{1,0}\ +=$ $Mds_{0,0}{}^*Nds_{1,0}\ +$ $Mds_{1,0}{}^*Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ $\downarrow$ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ $\downarrow$ $Nds_{0,1}$	$PdValue_{0,1}\ +=$ $Mds_{0,1}{}^*Nds_{0,0}\ +$ $Mds_{11}{}^*Nds_{0,1}$	$\mathbf{Md_{2,1}}$ $\downarrow$ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ $\downarrow$ $Nds_{0,1}$	$PdValue_{0,1}\ +=$ $Mds_{0,1}{}^*Nds_{0,0}\ +$ $Mds_{1,1}{}^*Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ $\downarrow$ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ $\downarrow$ $Nds_{1,1}$	$PdValue_{1,1}\ +=$ $Mds_{0,1}{}^*Nds_{1,0}\ +$ $Mds_{1,1}{}^*Nds_{1,1}$	$\mathbf{Md_{3,1}}$ $\downarrow$ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ $\downarrow$ $Nds_{1,1}$	$PdValue_{1,1}\ +=$ $Mds_{0,1}{}^*Nds_{1,0}\ +$ $Mds_{1,1}{}^*Nds_{1,1}$

time ⟶

FIGURE 5.6

Execution phases of a tiled matrix multiplication.

- The shared memory array for the Md elements is called Mds.
- The shared memory array for the Nd elements is called Nds.
- At the beginning of Phase 1, the four threads of block(0,0) collaboratively load a tile of Md into shared memory;
- thread(0,0) loads $Md_{0,0}$ into $Mds_{0,0}$, thread(1,0) loads $Md_{1,0}$ into $Mds_{1,0}$, thread(0,1) loads $Md_{0,1}$ into $Mds_{0,1}$, and thread(1,1) loads $Md_{1,1}$ into $Mds_{1,1}$.
- Look at thesecond column of Figu re 5.6. A tile of Nd is also loaded in a similar manner,as shown in the third column of Figure 5.6.

- After the two tiles of Md and Nd are loaded into the shared memory, these values are used in the calculation of the dot product.

- Note that each value in the shared memory is used twice; for example, the Md1,1 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 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. The reader should verify that the reduction is by a factor of N if the tiles are NN elements.

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- In the G80, each SM has 8K (= 8192) registers, which amounts to 128K(= 131,072) registers for the entire processor.

- This is a very large number, but it only allows each thread to use a very limited number of registers.

- Recall that each G80 SM can accommodate up to 768 threads. In order to fill this capacity, each thread can use only 8K/768 = 10 registers.

- If each thread uses 11 registers, the number of threads that can be on currently in each SM will be reduced. Such reduction is done at the block granularity.

- For example, if each block contains 256 threads, the number of threads will be reduced 256 at a time; thus, the next lower number of threads from 768 would be 512, a 1/3 reduction of threads that can simultaneously reside in each SM.

- This can greatly reduce the number of warps available for scheduling, thus reducing the processor's ability to find useful work in the presence of long-latency operations.