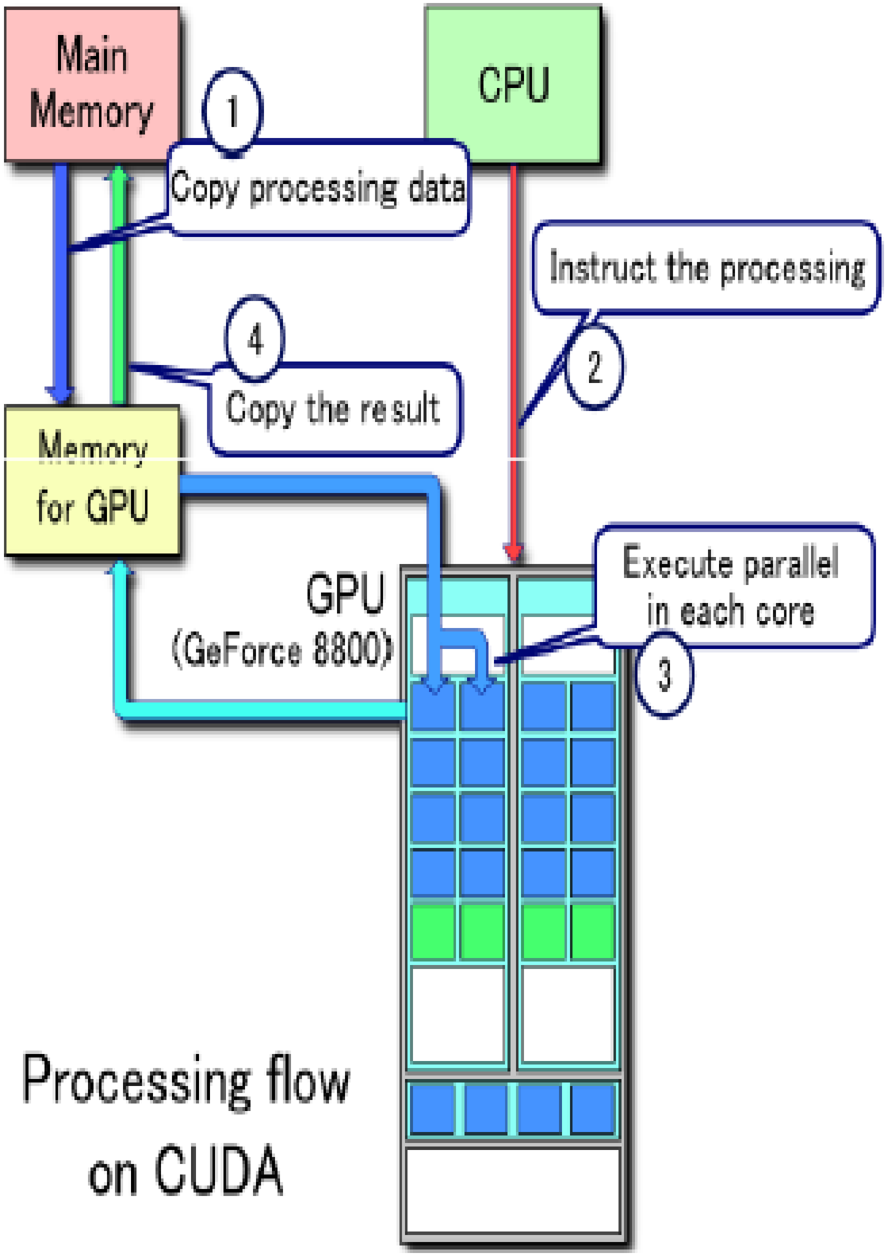
Introduction to CUDA

compiled by

Dr. N. Gopalakrishna Kini
Prof., Dept of CSE
MIT, Manipal

- ❑ Copy data from main memory to GPU memory
- ❑ CPU instructs the process to GPU
- ❑ GPU executes the compute in parallel on each core
- ❑ Copy the result from GPU memory to main memory



Processing flow on CUDA

What is Heterogeneous computing

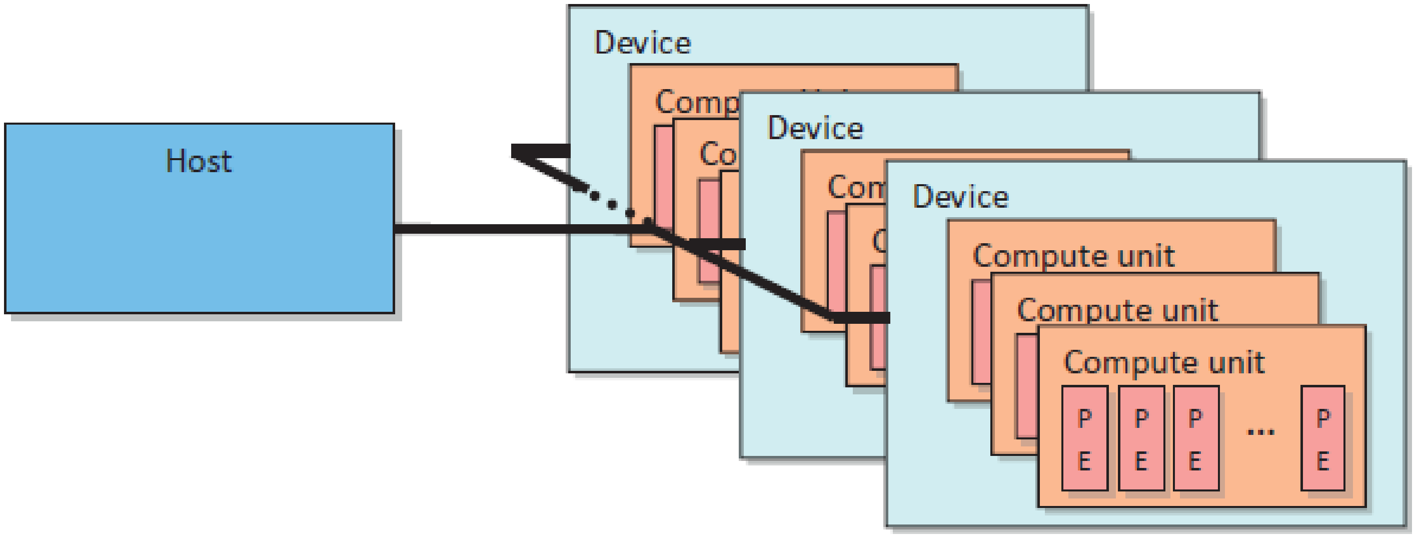
- **Heterogeneous computing** systems refer to electronic systems…
- A computational unit could be a GPP, a special-purpose processor or GPU, a co-processor, or FPGA.
- Another term sometimes seen for this type of computing is “Hybrid computing“
- GPU programs are called *kernels.*

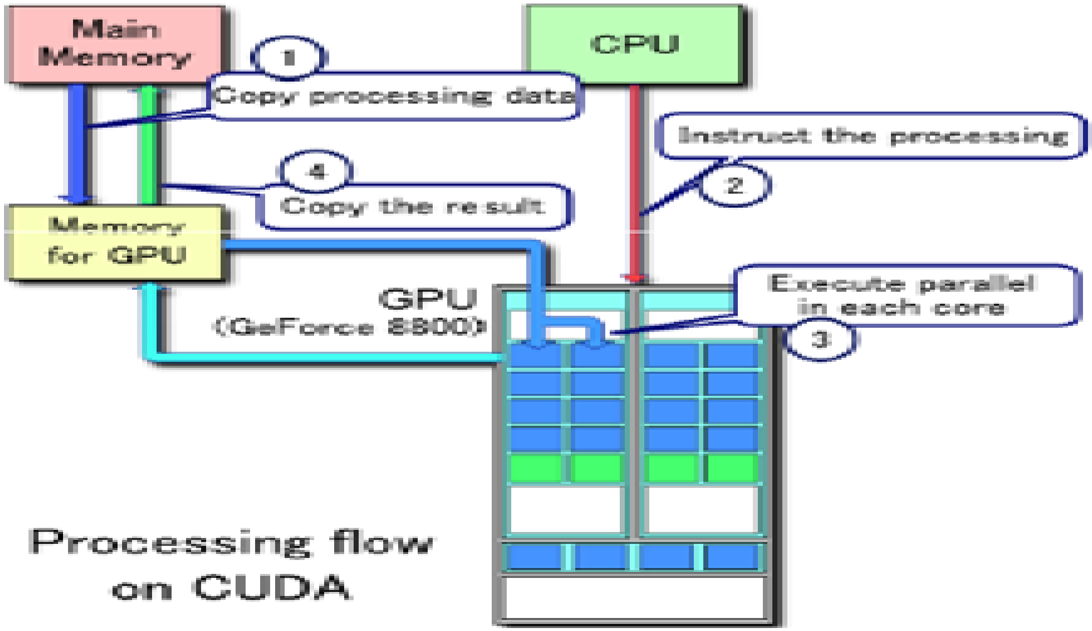
What is GPU Computing?

GPU computing…

Due to their massive-parallel architecture, GPU enables the computationally intensive assignments.

This is why GPU computing has enormous potential.





Processing flow on CUDA

INTRODUCTION

Computing system consists of a host, CPU, and one or more devices, which are massively parallel processors equipped with a large number of arithmetic execution units.

The Compute Unified Device Architecture (CUDA) devices accelerate the execution of these applications by harvesting a large amount of data parallelism.

Data parallelism plays such an important role in CUDA.

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.

The host code is straight ANSI C code; it is further compiled with the host’s standard C compiler and runs as an ordinary CPU.

The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures.

The device code is compiled by the nvcc and executed on a GPU device.

In situations where no device is available , one can also choose to execute kernels on a CPU using the emulation features in CUDA software development kit (SDK) or the MCUDA tool.

The kernel functions (or, simply kernels) typically generate a large number of threads to exploit data parallelism.

The execution starts with host (CPU) execution.
When a kernel function is invoked, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of rich data parallelism.
All the threads that are generated by a kernel during an invocation are collectively called a grid.
When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

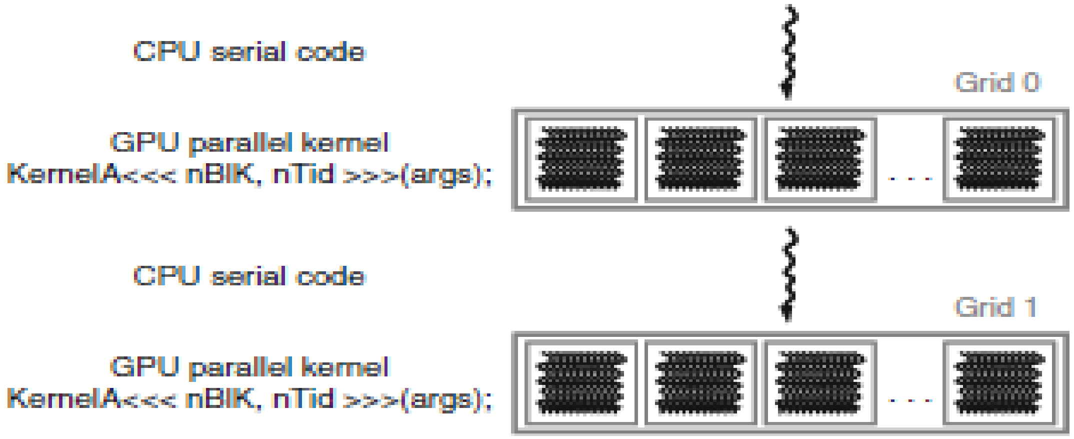


FIGURE 3.2

Execution of a CUDA program.

As in OpenCL
CPU is the host & its memory is the host memory
GPU is the device & its memory is device memory.

Serial code will be run on host & llel code will on device.

1. Copy data from host memory to device memory.
2. Load device program and execute, caching data on chip for performance.
3. Copy result from device memory to host memory.

```
#include "cuda_runtime.h"
#include “stdio.h"

_ _global_ _ void add(int *a, int *b, int *c)
{
   *c = *a + *b;
}

int main(void)
{
   int a, b, c;               // host copies of variables a, b & c
   int *d_a, *d_b, *d_c;     // device copies of variables a, b & c
   int size = sizeof(int);
```

```
// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Setup input values
a = 3;
b = 5;

// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

Explanation:

-add is the kernel function which runs on device.

-cudaMalloc ((void **)&d_a, size);

cudaMalloc will allocate memory on GPU of size bytes given as second argument to variable passed as first argument.

-cudaMemcpy (Destination, Source, Size, Direction);

-cudaMemcpy (d_a, &a, size, cudaMemcpyHostToDevice);

-cudaMemcpy (&c, d_c, size, cudaMemcpyDeviceToHost);

cudaMemcpy copies the variables from host to device or device to host based on the direction which is either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Value of size bytes long is copied from source to destination.

Explanation:

cudaFree frees the memory allocated by cudaMalloc.

The add function is called like this

add<<<1,1>>>(d_a, d_b, d_c)

The add is followed by three angular brackets,
then the number of blocks, threads per block
then corresponding closing angular brackets
then how many arguments the function add takes is enclosed within parenthesis.

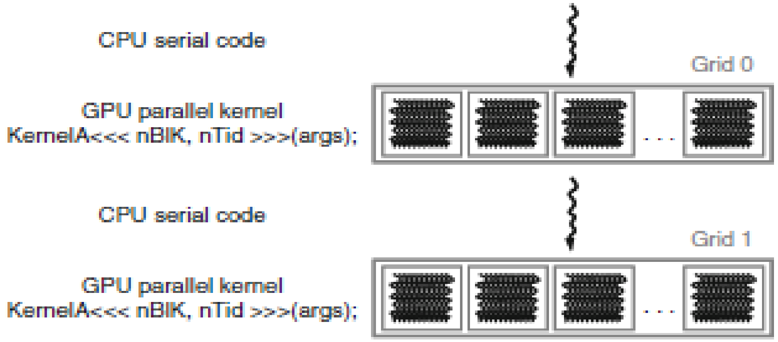


FIGURE 3.2

Execution of a CUDA program.

Explanation:

If you want to add N elements, you can achieve it in two ways either having N blocks as

That is pass an array with following function calls

```
add<<< N,1>>> (d_a, d_b, d_c)
```

or having N threads as

```
add<<< 1, N>>> (d_a,d_b,d_c)
```

Unlike OpenCL you can pass 2D array to a CUDA kernel.

Write a CUDA program to add corresponding elements of two arrays.

```
#include <iostream>
#include <cuda_runtime.h>

_ _ global _ _ void addArrays(int *a, int *b, int *c, int size)
{
    int tid = threadIdx.x;

    if (tid < size)
   {
        c[tid] = a[tid] + b[tid];
    }
}

int main() {
    int size = 10; // Size of the arrays
    int *h_a, *h_b, *h_c; // Host arrays
    int *d_a, *d_b, *d_c; // Device arrays
```

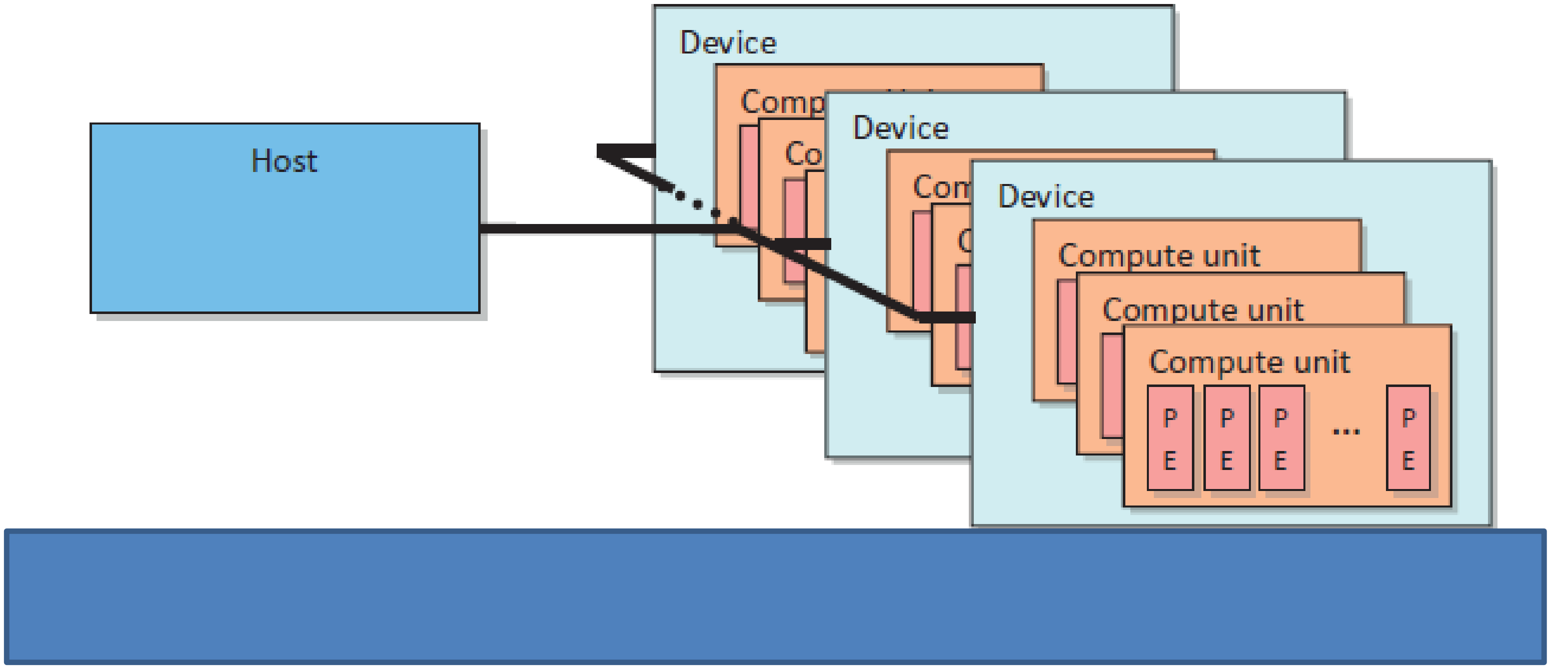
```
// Initialize host arrays
for (int i = 0; i < size; ++i) {
    h_a[i] = i;
    h_b[i] = 2 * i;
}

// Allocate memory for device arrays
cudaMalloc((void**)&d_a, size * sizeof(int));
cudaMalloc((void**)&d_b, size * sizeof(int));
cudaMalloc((void**)&d_c, size * sizeof(int));

// Copy data from host to device
cudaMemcpy(d_a, h_a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size * sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel with a single block of threads
addArrays<<<1, size>>>(d_a, d_b, d_c, size);

// Copy result back from device to host
cudaMemcpy(h_c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    // Print the result
    cout << "Result: ";
    for (int i = 0; i < size; ++i) {
        cout << h_c[i] << " ";
    }

    // Clean up
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

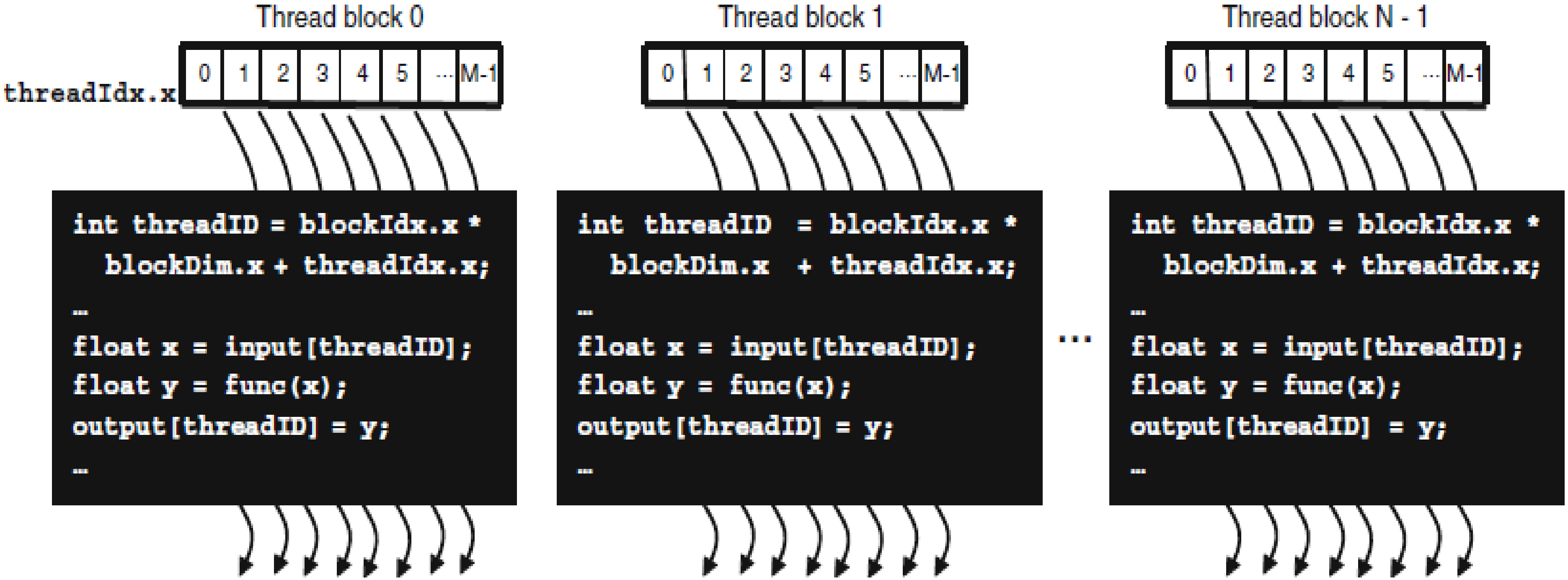
Specifying Grid/Block structure

- The programmer need to provide each kernel call with:
  - Number of blocks in each dimension
  - Threads per block in each dimension
    - **myKernel<<< B, T >>>(arg1, …);**

- **B** – a structure that defines the number of blocks in grid in each dimension (1D or 2D).
- **T** – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).
- B and T are of type dim3 (uint3).


Host
Device
Comp
Co
Device
Com
C
Device
Compute unit
Compute unit
Compute unit
P E
P E
P E
...
P E

CUDA THREAD ORGANIZATION (Contd..)



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int  threadID  = blockIdx.x *
  blockDim.x  + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID]  = y;
…
```

FIGURE 4.1

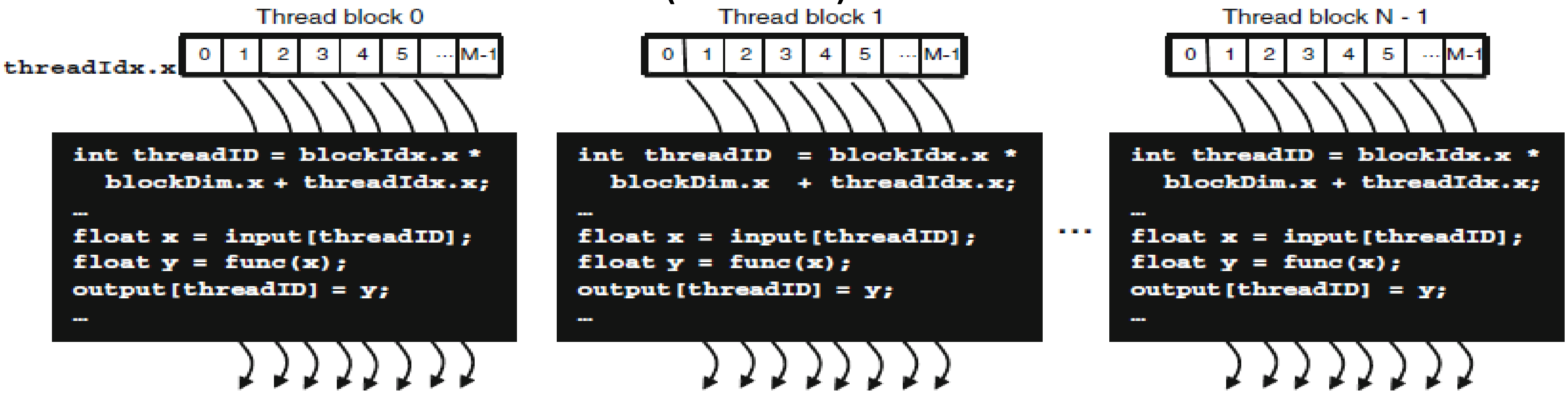
Overview of CUDA thread organization.

Fig. is a simple example of CUDA thread organization.

The grid consists of N blocks, each with a blockIdx.x value that ranges from 0 to N – 1.

Each block, further, consists of M threads, each thread with a threadIdx.x value that ranges from 0 to M – 1.

CUDA THREAD ORGANIZATION (Contd..)



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID  = blockIdx.x *
  blockDim.x  + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID]  = y;
…
```

FIGURE 4.1

Overview of CUDA thread organization.

All blocks at the grid level are organized as a one-dimensional (1D) array.

All threads within each block are also organized as a 1D array.

A grid has a total of (N*M) number of threads.

A thread in the kernel code shown uses

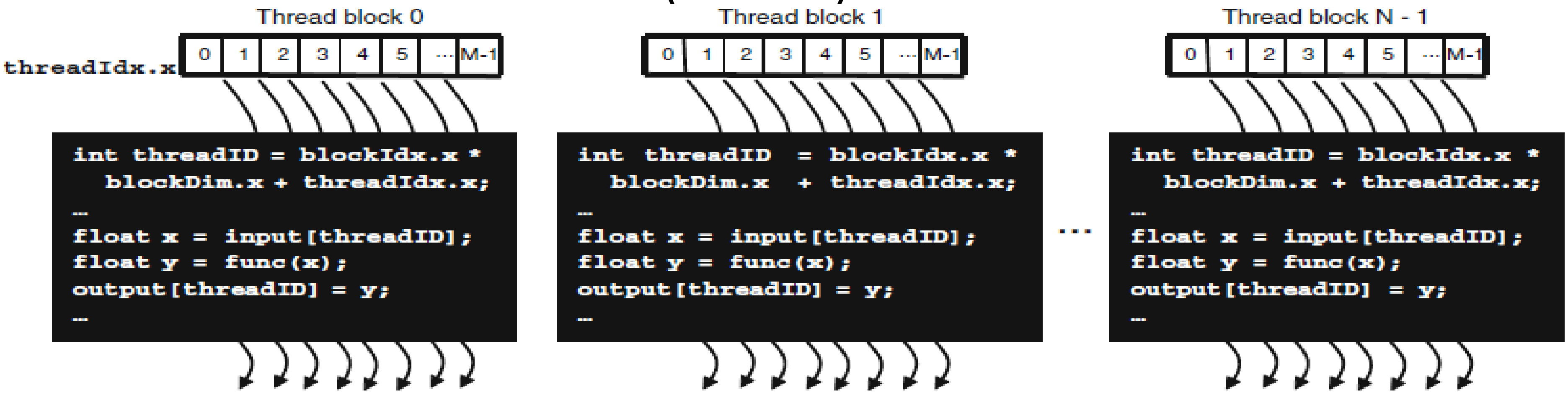
threadID = blockIdx.x * blockDim.x + threadIdx.x

to identify the part of the input data to read from and the part of the output data 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.

CUDA THREAD ORGANIZATION (Contd..)



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

FIGURE 4.1

Overview of CUDA thread organization.

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

Here, access to gridDim in the kernel returns 128. (dim3 dimGrid(size))

Here, access to blockDim in the kernel returns 32. (dim3 dimBlock(size))

There are a total of 128*32 = 4096 threads in the grid.

Thread 3 of Block 0 has a threadID value of 0*32 + 3 = 3.

Thread 3 of Block 5 has a threadID value of 5*32 + 3 = 163.

Thread 15 of Block 102 has a threadID value of 102*32 +15 =3279.

Note that each one of the 4096 threads has its own unique threaded value.

CUDA THREAD ORGANIZATION (Contd..)

If we assume both arrays are declared with 4096 elements, then each thread will take one of the input elements and produce one of the output elements.

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 det by the execution configuration provided at kernel launch.

The first parameter of the exec config specifies the dimensions of the grid in terms of number of blocks.

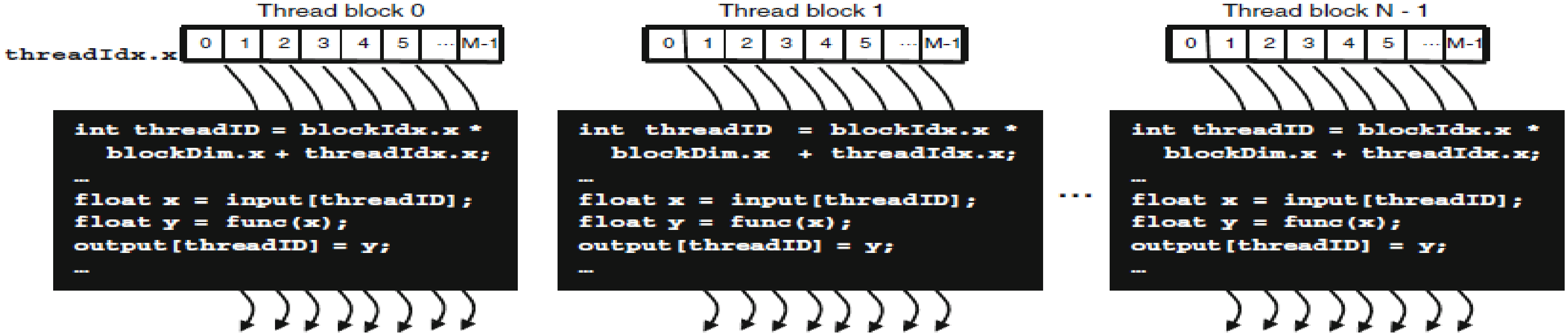
The second specifies the dimensions of each block in terms of number of threads.

Each such parameter is a dim3 type, which is essentially a C struct with three unsigned integer fields: x, y, and z.

As grids are 2D arrays of block dimensions, the third field of the grid dimension is ignored; it should be set to 1 for clarity.

CUDA THREAD ORGANIZATION (Contd..)

The following host code can be used to launch the kernel whose organization is shown below:



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID  = blockIdx.x *
  blockDim.x  + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
…
float x = input[threadID];
float y = func(x);
output[threadID]  = y;
…
```

FIGURE 4.1

Overview of CUDA thread organization.

dim3 dimGrid(128, 1, 1);

dim3 dimBlock(32, 1, 1);

KernelFunction<<<dimGrid, dimBlock>>>(. . .);

Because the grid and the blocks are 1D arrays, only the first dimension of dimBlock and dimGrid are used. The other dimensions are set to 1. The third statement is the actual kernel launch. The exec config parameters are between <<< and >>>.

CUDA THREAD ORGANIZATION (Contd..)

Note that scalar values can also be used for the exec config parameters if a grid or block has only one dimension;

Eg., the same grid can be launched with one statement:

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

The values of gridDim.x and gridDim.y can range from 0 to 65,535.

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.

CUDA THREAD ORGANIZATION (Contd..)

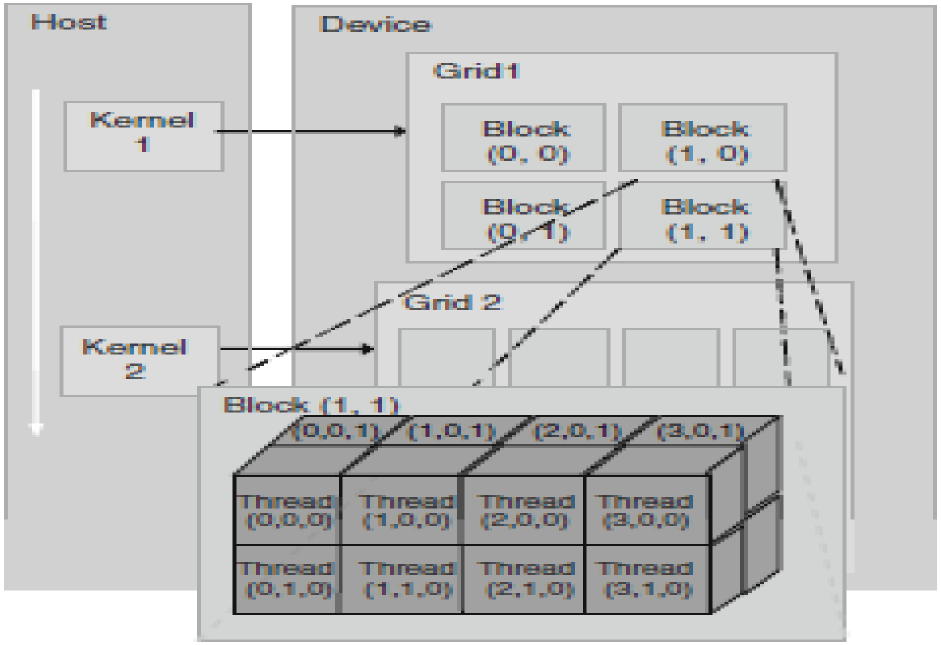


Fig. shows a small 2D grid that is launched with the following host code:

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

The grid consists of four blocks organized into a 2 X 2 array.

Each block is labeled with (blockIdx.x, blockIdx.y); for example, Block(1,0) has blockIdx.x = 1 and blockIdx.y = 0.

CUDA THREAD ORGANIZATION (Contd..)

In general, 2 D blocks are organized into 3D arrays of threads.

All blocks in a grid have the same dimensions.

Each threadIdx consists of three components:

the x coordinate threadIdx.x, threadIdx.y, and threadIdx.z.

The total size of a block is limited to 512 threads,

with flexibility in distributing these elements into the three dimensions.

Eg. (512, 1, 1), (8, 16, 2), and (16, 16, 2) are all allowable blockDim values, but (32, 32, 1) is not allowable because the total number of threads would be 1024.

Specifying Grid/Block structure

- The programmer need to provide each kernel call with:
  - Number of blocks in each dimension
  - Threads per block in each dimension
    - **myKernel<<< B, T >>>(arg1, …);**

- **B** – a structure that defines the number of blocks in grid in each dimension (1D or 2D).

- **T** – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).
- B and T are of type dim3 (uint3).

1-D grid and/or 1-D blocks

- For 1-D structure, one can use an integer for each of B and T in:
    - **myKernel<<< B, T >>>(arg1, …);**
- **B** – *An integer would define a 1D grid of that size*
- **T** –*An integer would define a 1D block of that size*
    - **myKernel<<< 1, 100 >>>(arg1, …);**

- Grids can be 2D and blocks can be 2D or 3D

Compute global 1-D thread ID

- dim3
- **threadIdx.x** -- “thread index” within block in “x” dimension
- **blockIdx.x** -- “block index” within grid in “x” dimension
- **blockDim.x** -- “block dimension” in “x” dimension (i.e. number of threads in a block in the x dimension)
- Full global thread ID in x dimension can be computed by:

x = blockIdx.x * blockDim.x + threadIdx.x;

In the mat mul eg, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix.

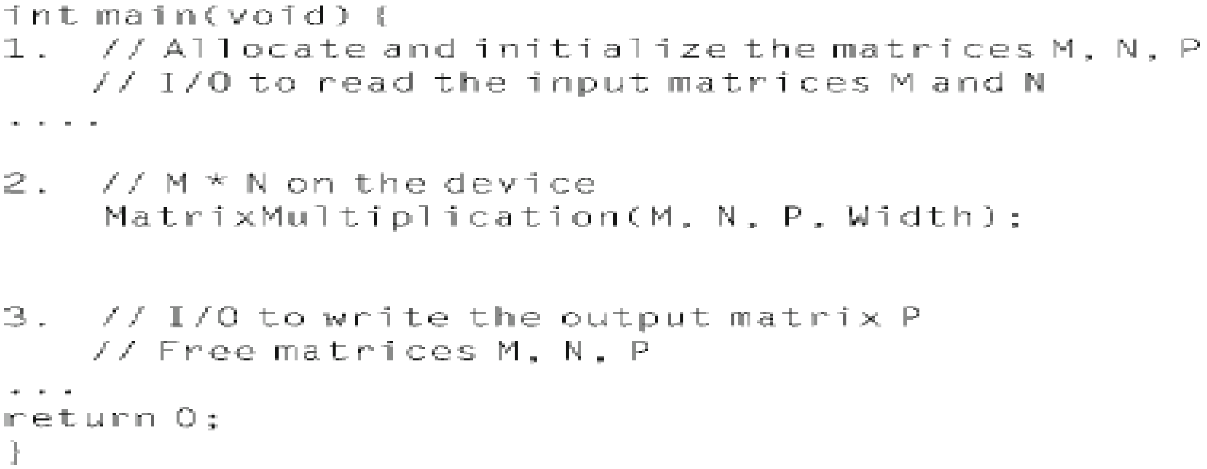
CUDA threads are of much lighter weight than the CPU threads.

Due to efficient hardware support, CUDA programmers finds that these threads take very few cycles to generate and schedule.

This is in contrast with the CPU threads that typically require thousands of clock cycles.

A MATRIX–MATRIX MULTIPLICATION EXAMPLE

To illustrate the CUDA program structure, first consider a simple main function skeleton for the matrix multiplication.



```
int main(void) {
1.   // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.   // M * N on the device
     MatrixMultiplication(M, N, P, Width);


3.   // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

FIGURE 3.3

A simple main function for the matrix multiplication example.

In the pgm first it allocates M, N, and P matrices in the host memory.

Then it performs I/O read in M and N.

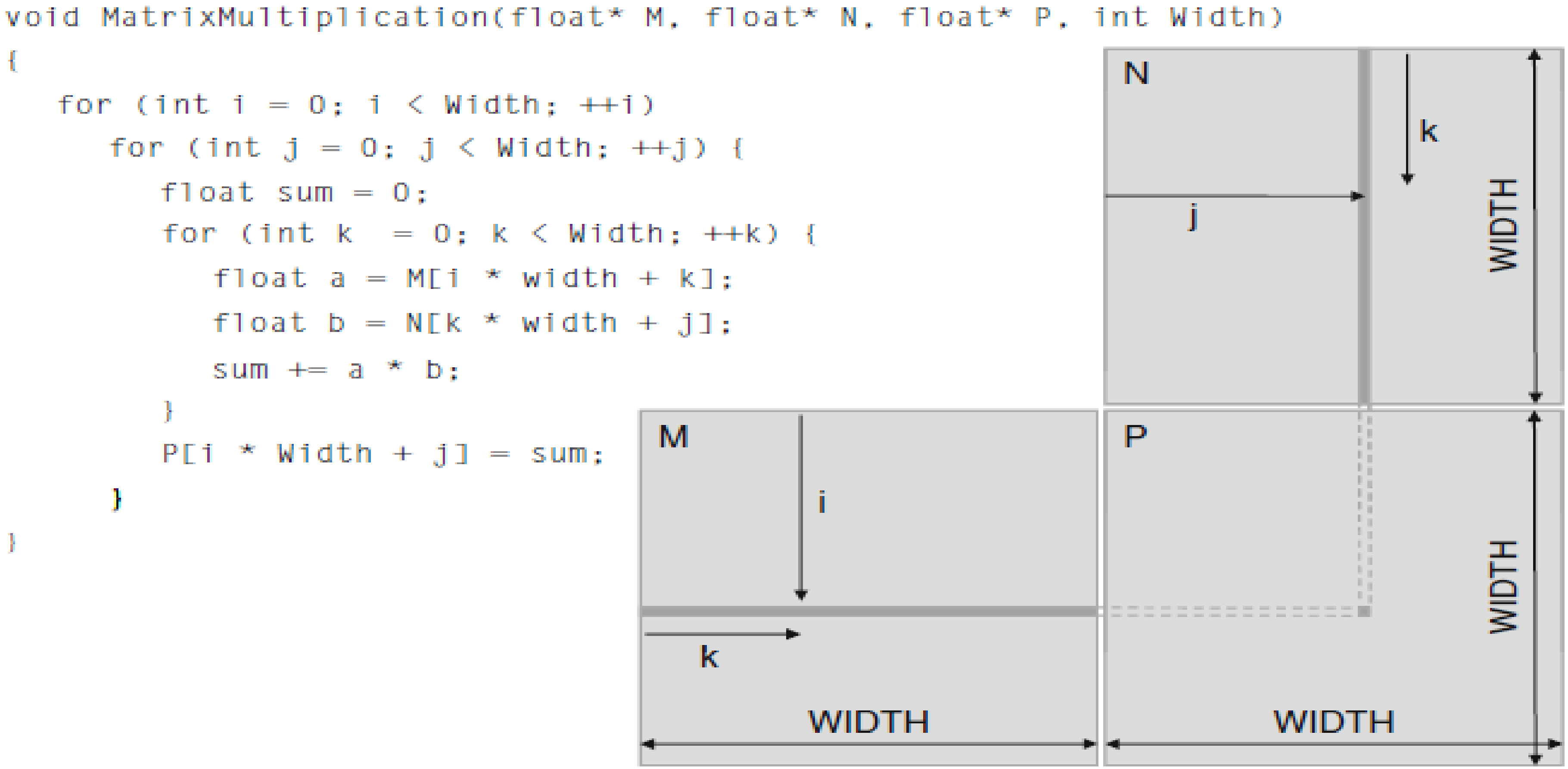
Part 3 performs I/O to write the product matrix P and to free all the allocated matrices.

Part 2 is the main focus of our interest.

A MATRIX–MATRIX MULTIPLICATION EXAMPLE (Contd..)

In PART 2 call to a function, MatrixMultiplication(), to perform mat mul on a device.

Before to know how to use a CUDA device to execute the mat mul fn, let us see a conventional CPU-only matrix multiplication function.



```
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 3.4

A simple matrix multiplication function with only host code.

A MATRIX–MATRIX MULTIPLICATION EXAMPLE (Contd..)

The MatrixMultiplication() fn impl a straightforward algo consisting of 3 loop levels.

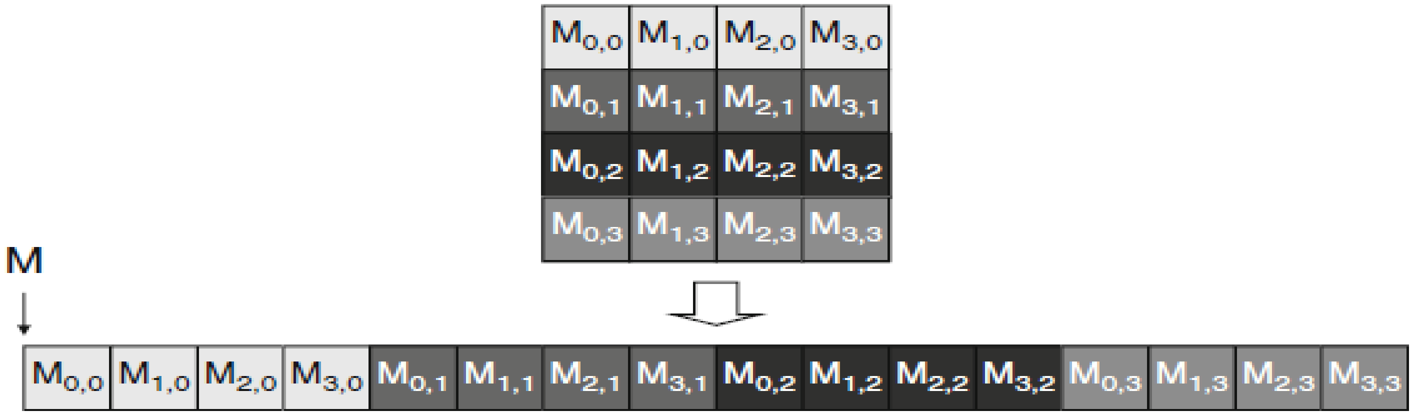
Innermost loop iterates over variable k and steps through one row of matrix M and one column of matrix N.

This loop calc a dot product of the row of M and the column of N to generate one element of P.

The index used for accessing the M matrix in the innermost loop is

(i*Width+k).

M matrix elements are placed into this linear addressed memory is done according to the row-major convention, as below:



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

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

FIGURE 3.5

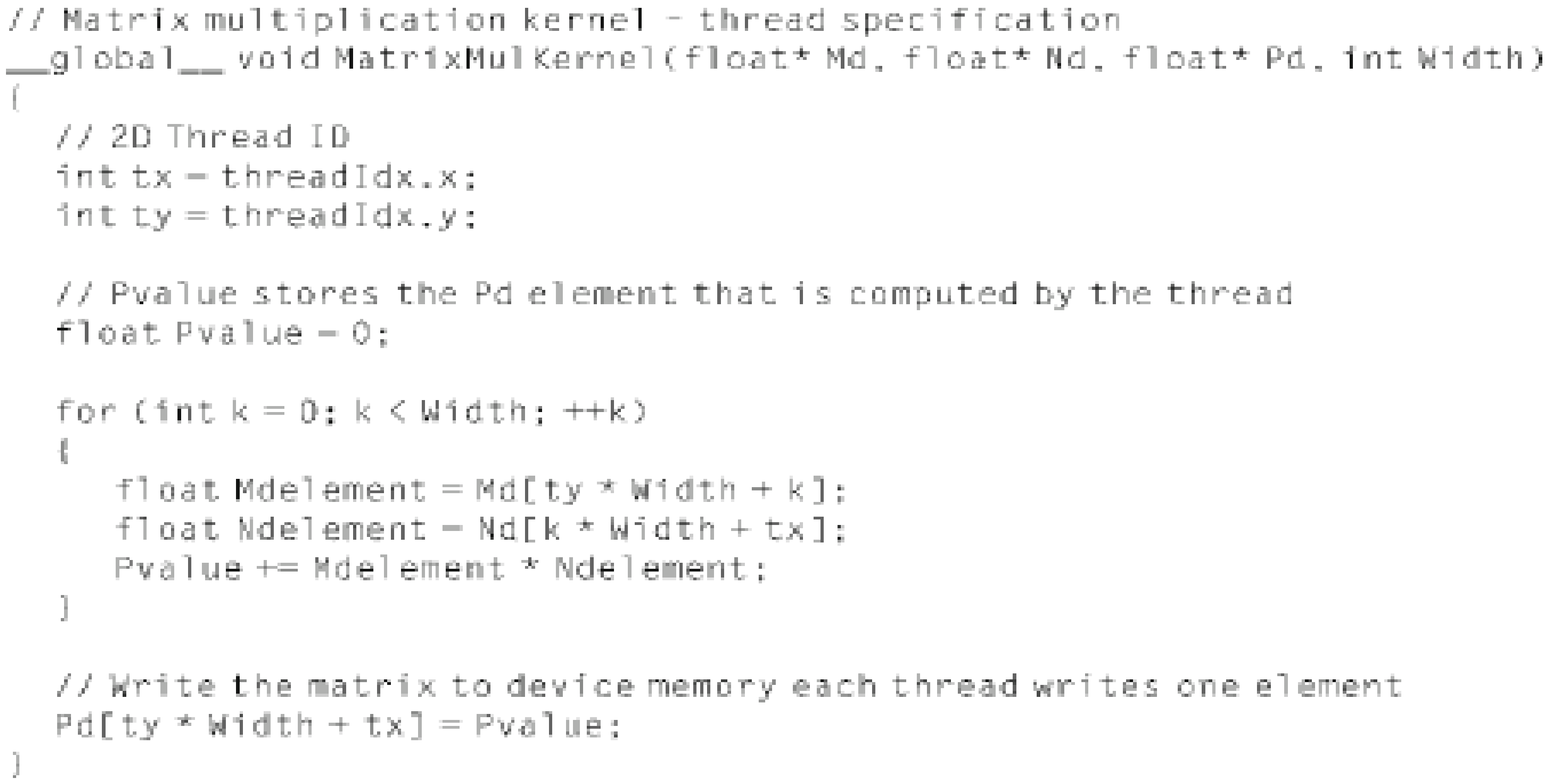
Placement of two-dimensional array elements into the linear address system memory.

KERNEL FUNCTIONS AND THREADING

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 single-program, multiple-data (SPMD) parallel programming.

The kernel function for matrix multiplication.

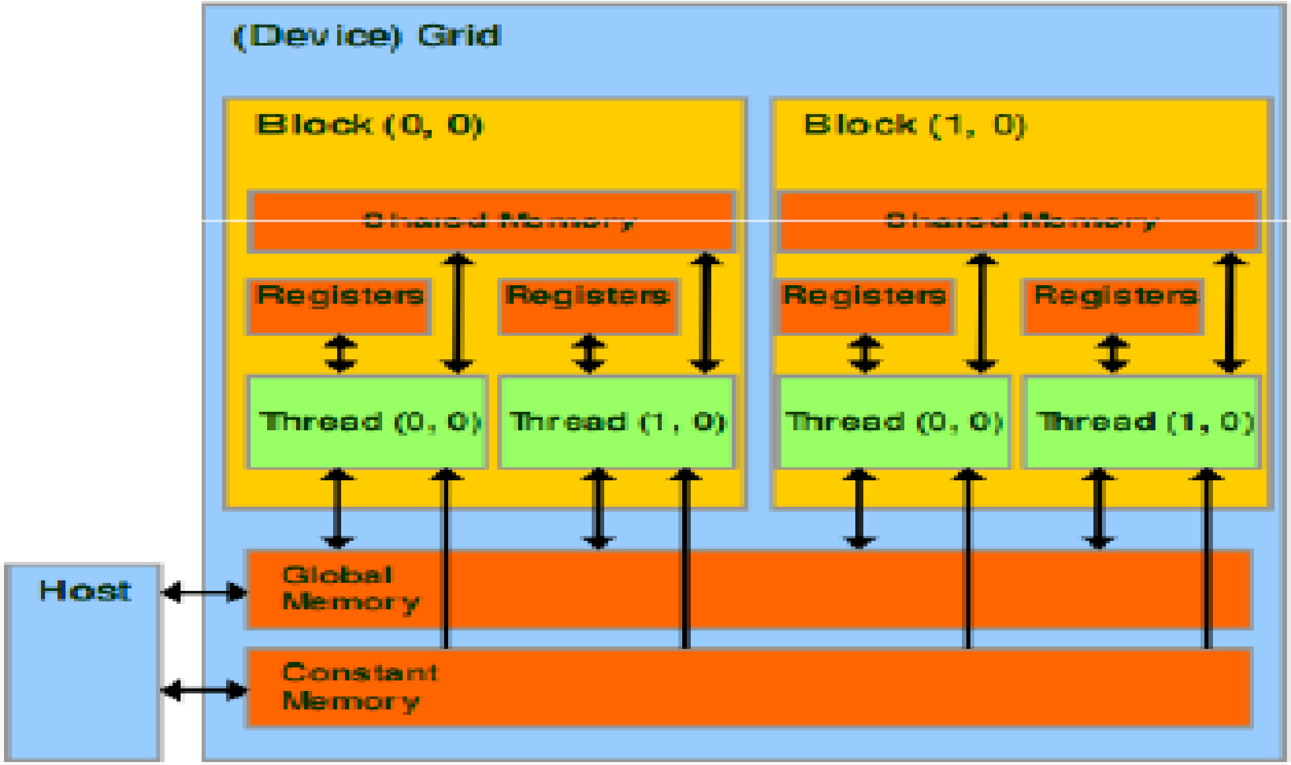


```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // 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;
}
```

FIGURE 3.11

The matrix multiplication kernel function.

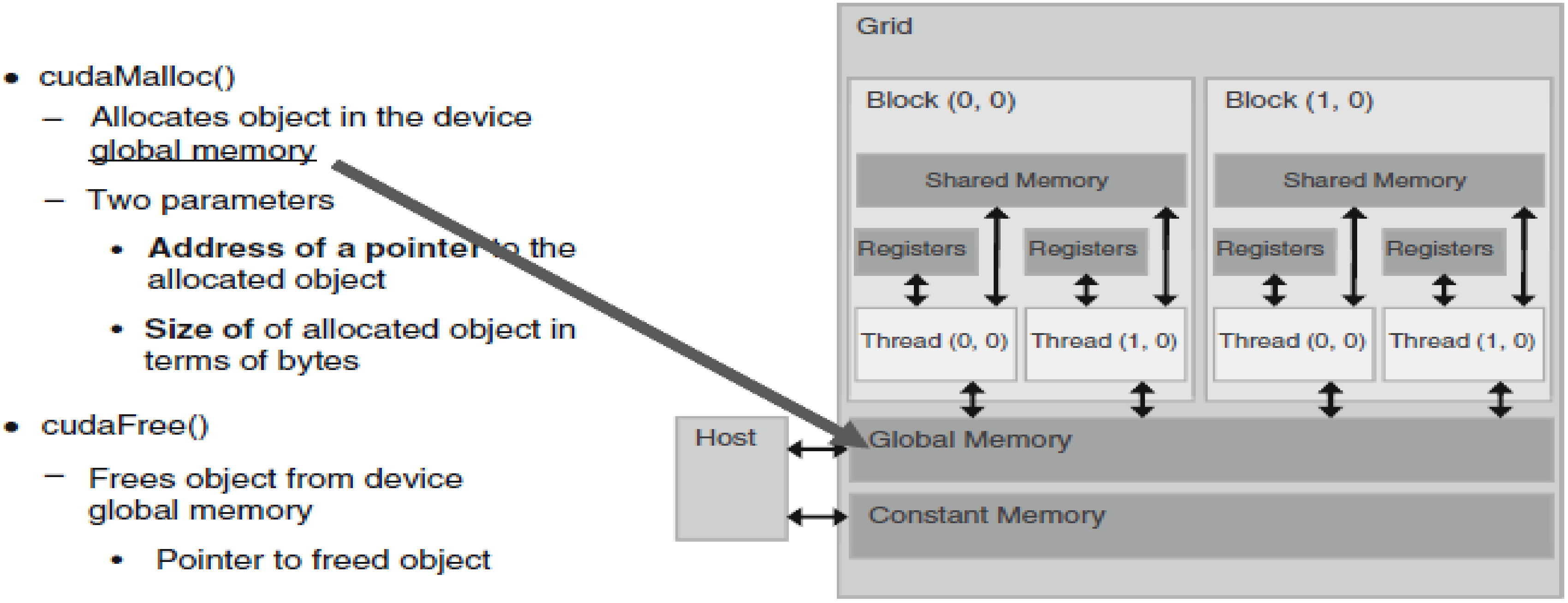
DEVICE MEMORIES AND DATA TRANSFER



Global memory and constant memory are the memories that the host code can transfer data to and from the device, as illustrated by the bidirectional arrows b/n these memories and the host.

Constant memory allows read-only access by the device code.

DEVICE MEMORIES AND DATA TRANSFER (Contd..)



- 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

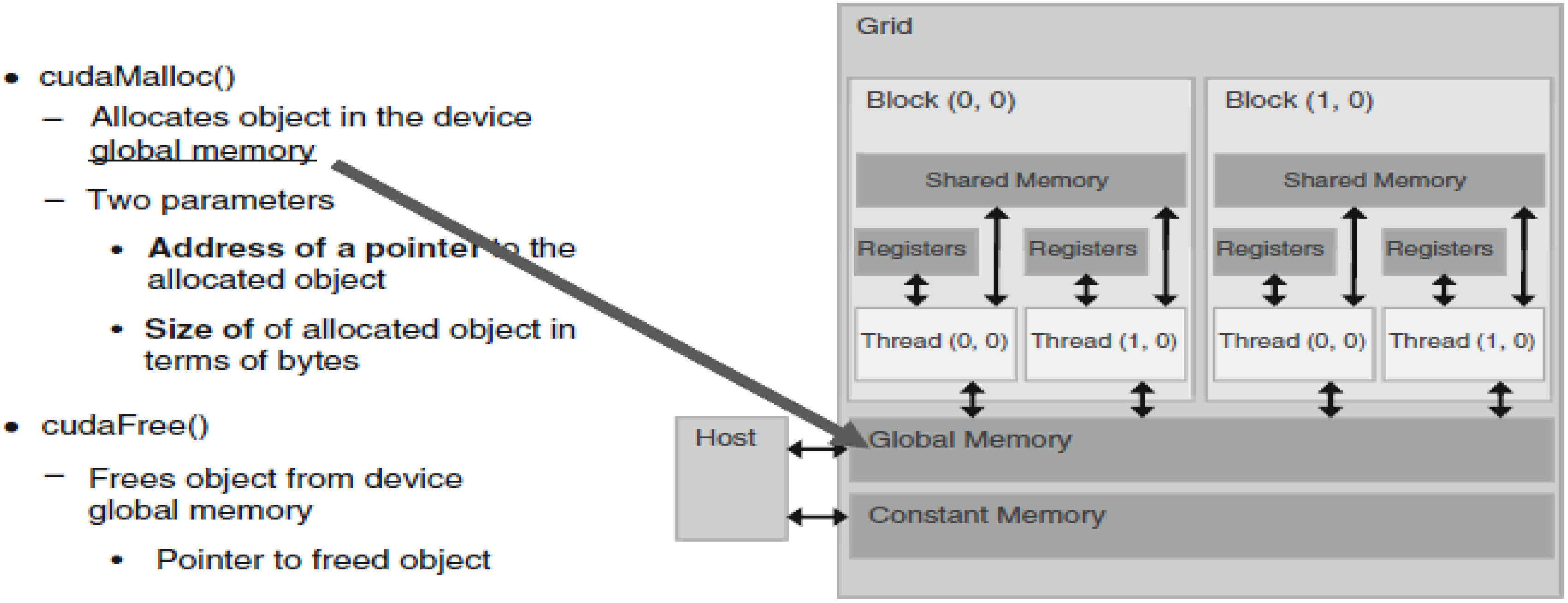
FIGURE 3.8

CUDA API functions for device global memory management.

Fig. shows the API functions for allocating and deallocating device global memory.

```
cudaMalloc ((void **)&d_a, size);

cudaFree(d_a);
```

DEVICE MEMORIES AND DATA TRANSFER (Contd..)



- 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.

The function cudaMalloc() can be called from the host code to allocate global memory for an object.

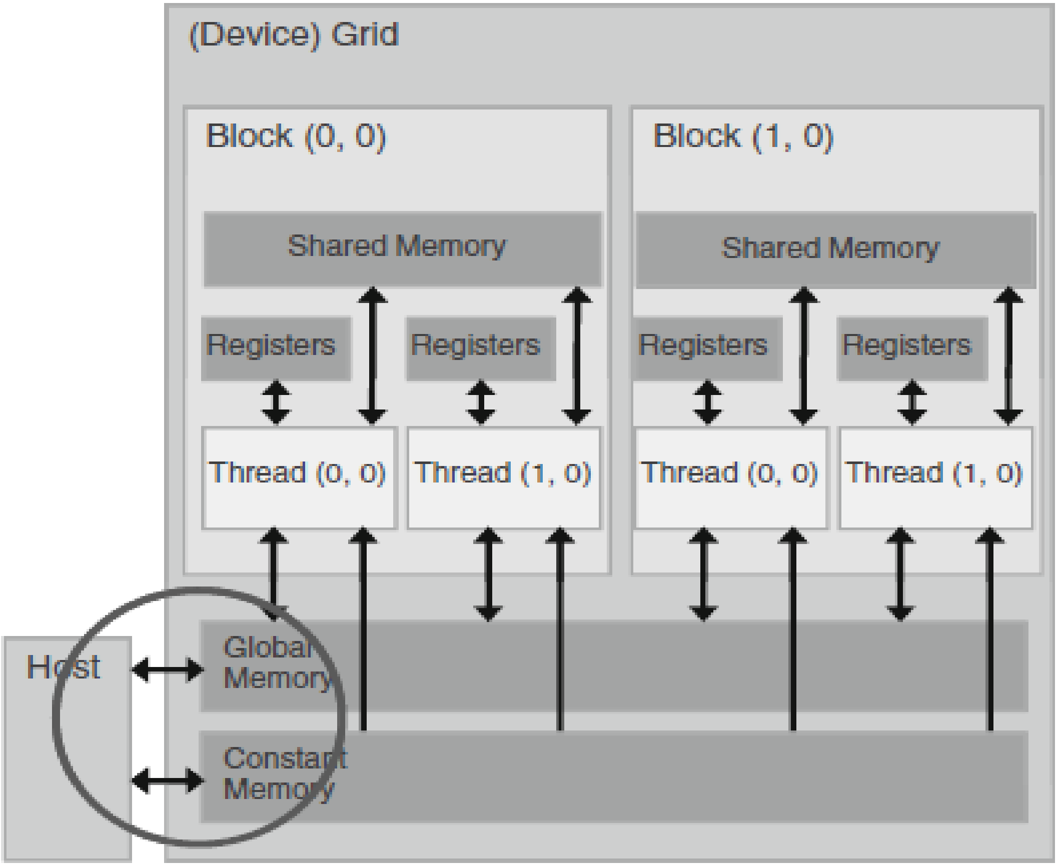
Striking IIlty b/n cudaMalloc() and the standard C runtime library malloc().

This is intentional. CUDA uses the standard C runtime library malloc() function to manage the host memory and adds cudaMalloc() as an extension to the C runtime library.

DEVICE MEMORIES AND DATA TRANSFER (Contd..)

Once a program has allocated device global memory for the data objects, it can request that data be transferred from host to device. This is accomplished by cudaMemcpy()

- cudaMemcpy()
  - **Memory** data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous



```
cudaMemcpy (d_a, &a, size, cudaMemcpyHostToDevice);
```

FIGURE 3.9

CUDA API functions for data transfer between memories.

- 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

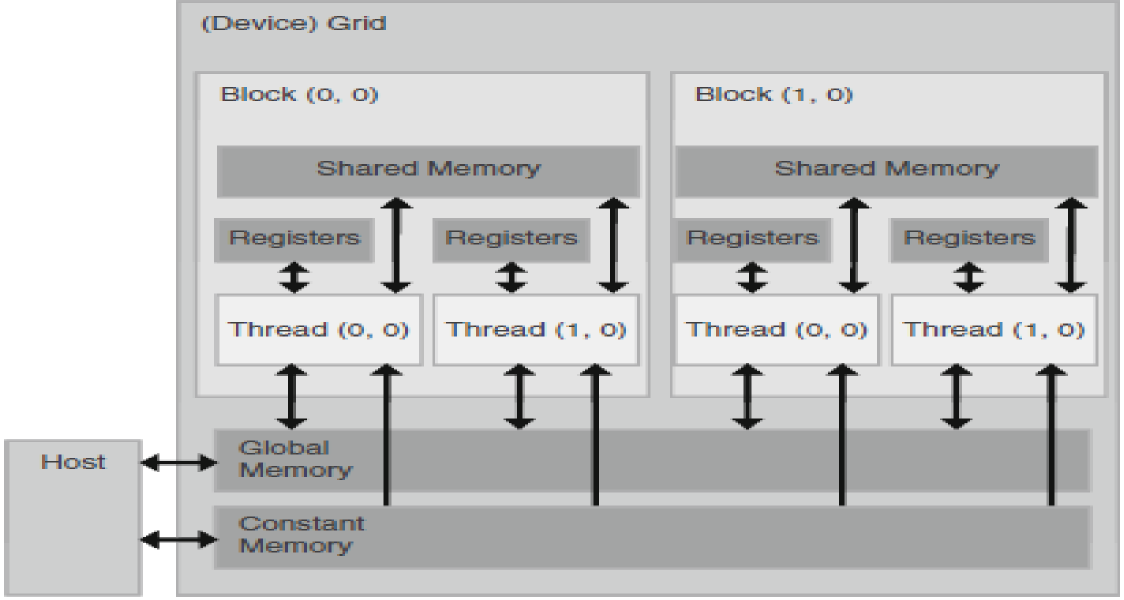


FIGURE 5.2

Overview of the CUDA device memory model.

CUDA DEVICE MEMORY TYPES (Contd..)

Global and constant memory can be written (W) and read (R) by the host by calling APIs.

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

Registers and shared memory are on-chip memories.

Variables that reside in these types of memory can be accessed at very high speed in a highly llel manner.

CUDA DEVICE MEMORY TYPES (Contd..)

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.

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.

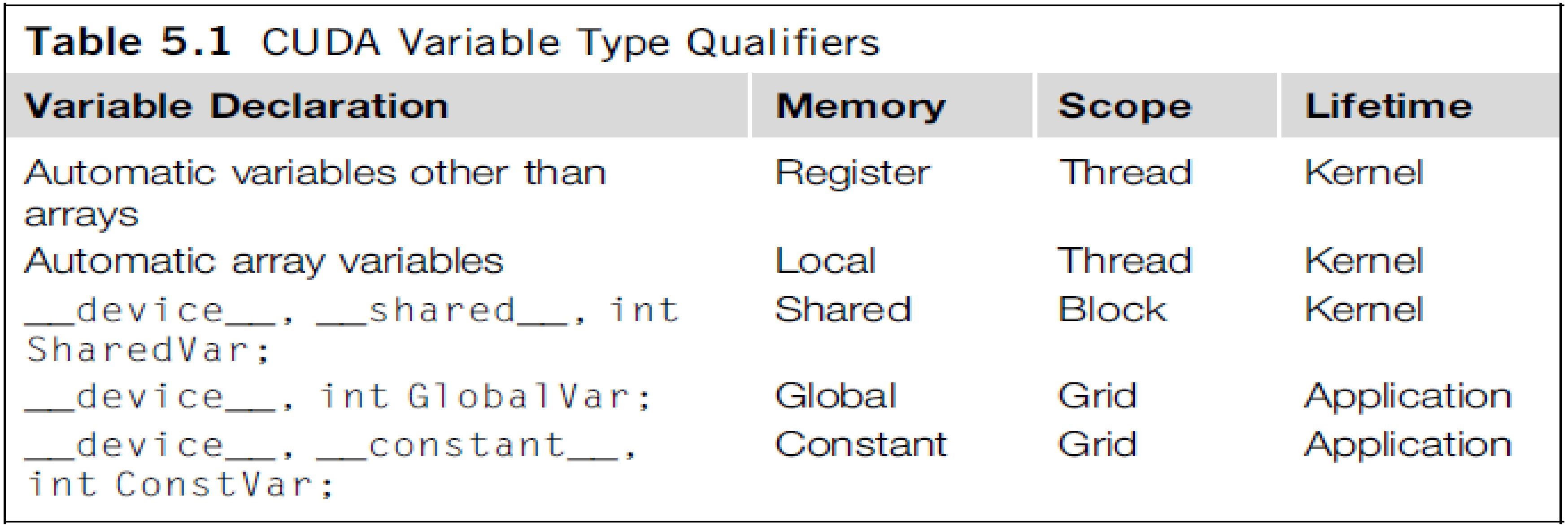


Table 5.1 CUDA Variable Type Qualifiers

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 (Contd..)

The scope of these arrays is, like automatic scalar variables, limited to individual threads.

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 declarations must reside within a kernel fn or a device function.

The scope of a shared variable is within a thread block; i.e., all threads in a block see the same version of a shared variable.

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.

When a kernel terminates its execution, the contents of its shared variables cease to exist.

Shared variables are an efficient means for threads within a block to collaborate with each other.

Accessing shared memory is extremely fast and highly parallel.

CUDA DEVICE MEMORY TYPES (Contd..)

CUDA programmers often use shared memory to hold the portion of global memory data that are heavily used in an execution phase of the kernel.

One may need to adjust the algorithms used.

CUDA DEVICE MEMORY TYPES (Contd..)

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.

With appropriate access patterns, accessing constant memory is extremely fast and all the threads can access in parallel.

Currently, the total size of constant variables in an application is limited at 64 Kbytes.

CUDA DEVICE MEMORY TYPES (Contd..)

A variable whose declaration is preceded only by the keyword _ _device_ _ is a global variable and will be placed in global memory.

Accesses to a global variable are slow; however, global variables are visible to all threads of all kernels.

Their contents also persist through the entire execution.

Hence, global variables can be used as a means for threads to collaborate across blocks.

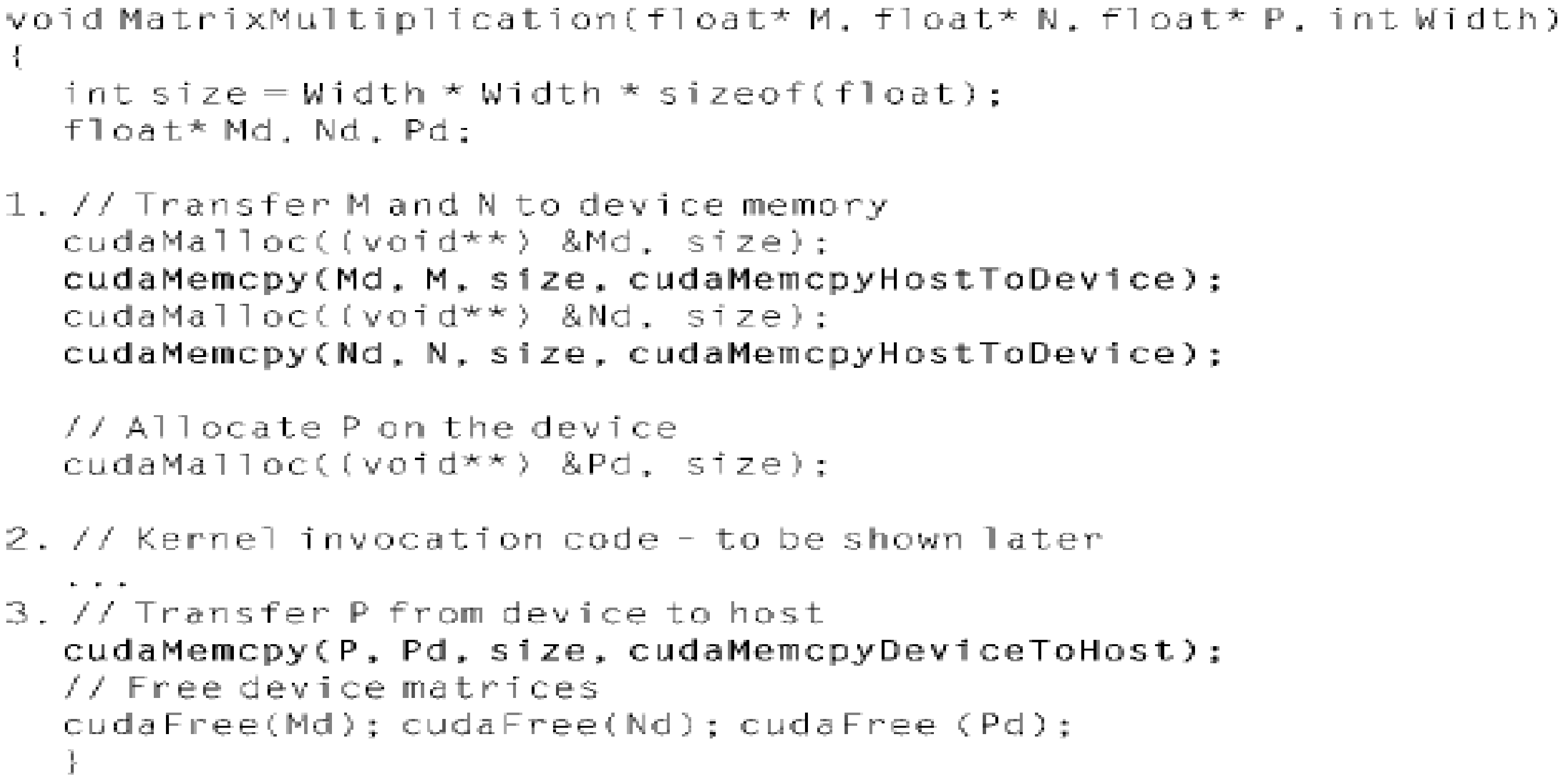
Global variables are often used to pass information from one kernel invocation to another kernel invocation.

..

DEVICE MEMORIES AND DATA TRANSFER (Contd..)

Note that cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost, are predefined constants of the CUDA programming environment.

The same API can be used to transfer data in both directions by properly ordering the source and destination pointers.



```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;

1. // Transfer M and N to device memory
   cudaMalloc((void**) &Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer P from device to host
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

FIGURE 3.10

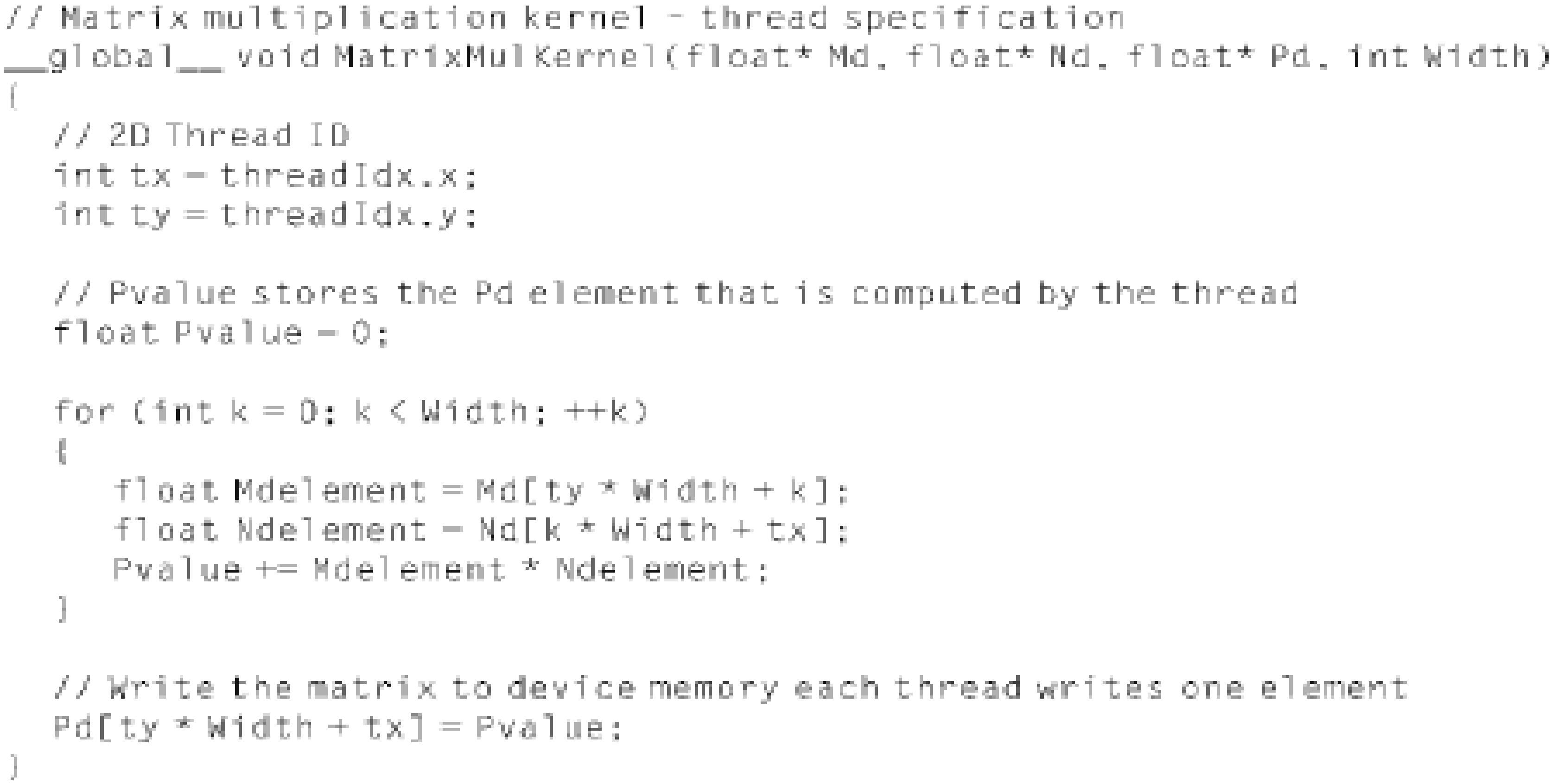
The revised MatrixMultiplication() function.

KERNEL FUNCTIONS AND THREADING

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 single-program, multiple-data (SPMD) parallel programming.

The kernel function for matrix multiplication.



```
// 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;
}
```

FIGURE 3.11

The matrix multiplication kemel function.

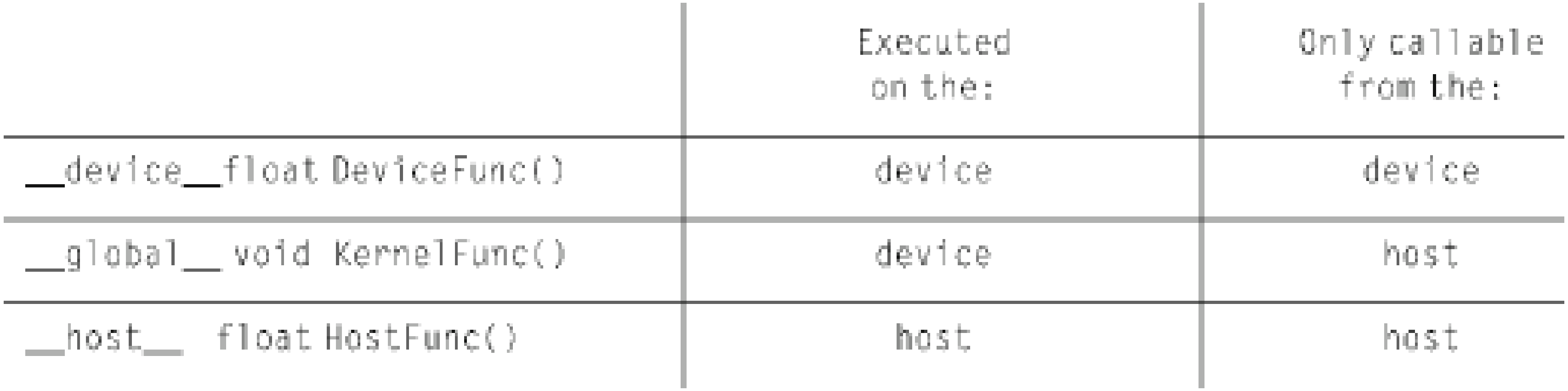
KERNEL FUNCTIONS AND THREADING (Contd..)

First, there is a CUDA-specific keyword “_ _global_ _” in front of the declaration of MatrixMulKernel().

-This keyword indicates that the function is a kernel

-it can be called from a host functions to generate a grid of threads on a device.

Besides _ _global_ _, there are two other keywords which can be used in front of a function declaration.



	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.

KERNEL FUNCTIONS AND THREADING (Contd..)

The _ _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.

The _ _host_ _keyword indicates that

-the function being declared is a CUDA host function.

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

If the functions do not have any of the CUDA keywords in their declaration, they will become host functions in a CUDA program.

KERNEL FUNCTIONS AND THREADING (Contd..)

Other keywords are

threadIdx.x and threadIdx.y

which refer to the thread indices of a thread.

All threads execute the same kernel code.

There needs a mechanism to allow threads to distinguish themselves and direct themselves toward the required part of the data structure that they are designated to work on.

These keywords identify predefined variables that allow a thread to access the data at runtime by identifying coordinates to the thread.

Different threads will see different values in their threadIdx.x and threadIdx.y variables.

KERNEL FUNCTIONS AND THREADING (Contd..)

Note that the coordinates x and y reflect a multidimensional organization for the threads.

Kernel function has only one loop, which corresponds to the innermost loop k.

Where the other two levels of outer loops go?

The answer is that the outer two loop levels are now replaced with the grid of threads.

The entire grid forms the equivalent of the two-level loop.

Each thread in the grid corresponds to one of the iterations of the outer two loops.

The loop variables i and j are now replaced with threadIdx.x and threadIdx.y.

Instead of having the loop increment the values of i and j, the CUDA threading hardware generates all of the threadIdx.x and threadIdx.y values for each thread.

In our kernel pgm, each thread uses its threadIdx.x and threadIdx.y to identify the row of Md and the col of Nd to perform the dot product.

KERNEL FUNCTIONS AND THREADING (Contd..)

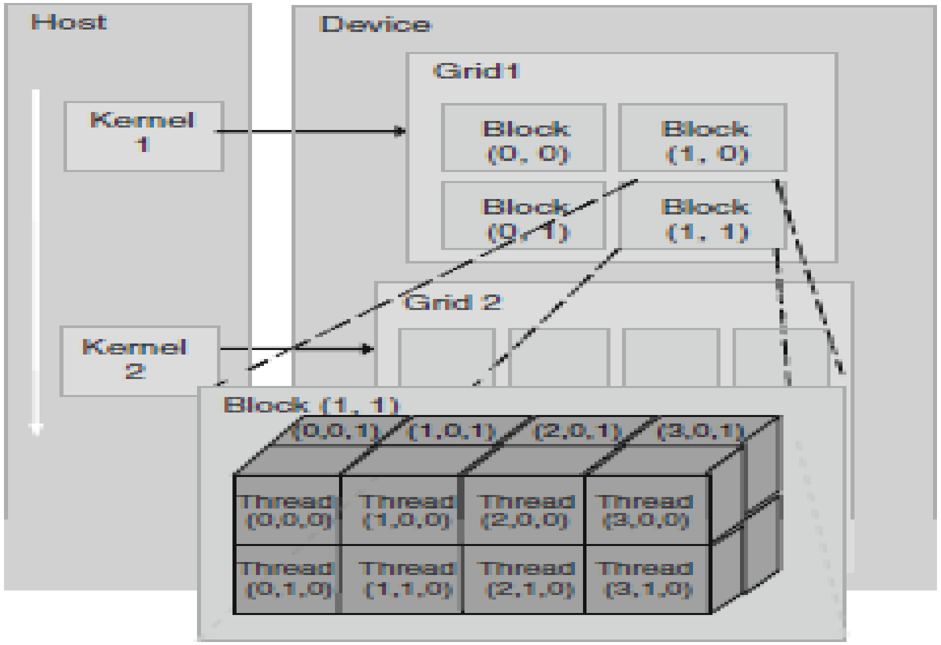
Here, we assigned threadIdx.x to the variable tx and threadIdx.y to variable ty.

Each thread uses its threadIdx.x and threadIdx.y values to select the Pd element that it is responsible for.

For eg, Thread$_{2,3}$ will perform a dot product between column 2 of Nd and row 3 of Md and write the result into element (3, 2) of Pd.

This way, the threads collectively generate all the elements of the Pd matrix.

When a kernel is invoked, or launched, it is executed as grid of parallel threads.



KERNEL FUNCTIONS AND THREADING (Contd..)

In Fig., the launch of Kernel 1 creates Grid 1.

Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation.

Each element of a large array might be computed in a separate thread.

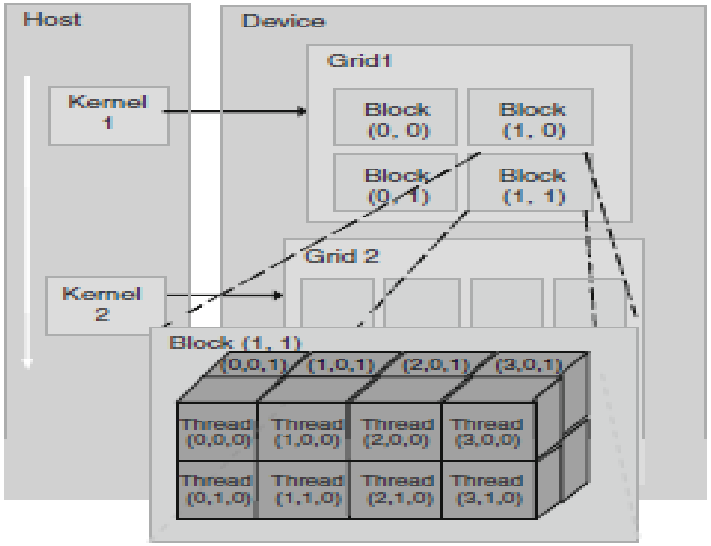
Threads in a grid are organized into a two-level hierarchy, as in Fig.

At the top level, each grid consists of one or more thread blocks.

All blocks in a grid have the same number of threads.

In Fig., Grid 1 is organized as a 2 x 2 array of 4 blocks.

Each block has a unique two-dimensional coordinate given by the CUDA specific keywords blockIdx.x and blockIdx.y.

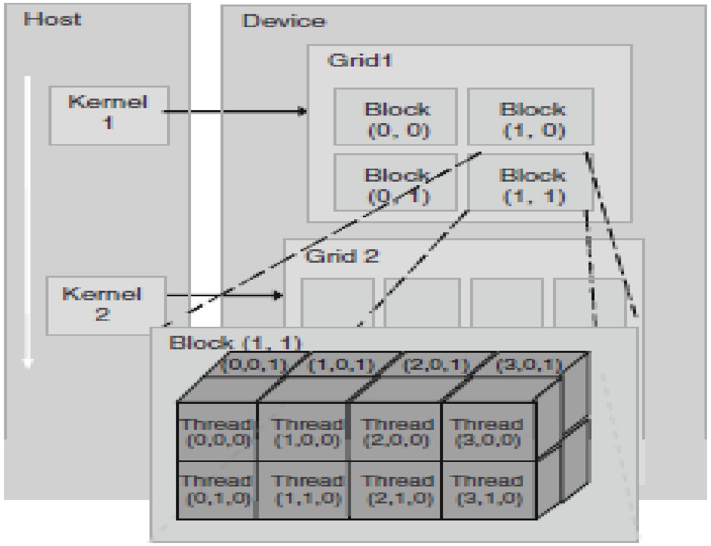


KERNEL FUNCTIONS AND THREADING (Contd..)

Each thread block is, in turn, organized as a three-dimensional array of threads with a total size of up to 512 threads.

The coordinates of threads in a block are uniquely defined by three thread indices: threadIdx.x, threadIdx.y, and threadIdx.z.

Not all applications will use all three dimensions of a thread block. In Figure, each thread block is organized into a 4 x 2 x 2 three-dimensional array of threads. This gives Grid 1 a total of 4 x 16 = 64 threads.



KERNEL FUNCTIONS AND THREADING (Contd..)

In our matrix multiplication example, a grid is invoked to compute the product matrix.

The code does not use any block index in accessing input and output data.

Threads with the same threadIdx values from different blocks would end up accessing the same input and output data elements.

As a result, the kernel can use only one thread block.

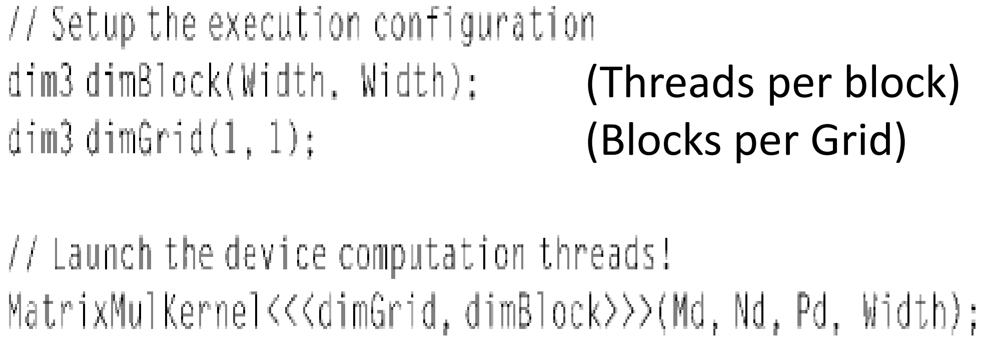
The threadIdx.x and threadIdx.y values are used to organize the block into a two-dimensional array of threads.

Because a thread block can have only up to 512 threads, and each thread calculates one element of the product matrix, the code can only calculate a product matrix of up to 512 elements.

As we explained before, the product matrix must have millions of elements in order to have a sufficient amount of data parallelism to benefit from execution on a device.

KERNEL FUNCTIONS AND THREADING (Contd..)

When the host code invokes a kernel, it sets the grid and thread block dimensions via execution configuration parameters.



```
// Setup the execution configuration
dim3 dimBlock(Width, Width);        (Threads per block)
dim3 dimGrid(1, 1);                 (Blocks per Grid)

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

FIGURE 3.14

Example of host code that launches a kernel.

This is illustrated above. Two struct variables of type dim3 are declared.

The first is for describing the configuration of blocks.

-which are defined as 16 x 16 groups of threads.

The second variable, dimGrid, describes the configuration of the grid.

-In this example only one (1 x 1) block in grid.

The final line of code invokes the kernel.

It provides the dimensions of the grid in terms of number of blocks and the dimensions of the blocks in terms of number of threads.

Some of the calculations for indexing the thread is given below.

1D grid of 1D blocks:

```
__device__int getGlobalID_1D_1D(){
                return blockIdx.x *blockDim.x + threadIdx.x;
            }
```

1D grid of 2D blocks

```
__device__int getGlobalID_1D_2D(){
        return blockIdx.x * blockDim.x * blockDim.y
                        + threadIdx.y * blockDim.x + threadIdx.x;
}
```

1D grid of 3D blocks

```
__device__int getGlobalID_1D_3D(){
        return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
                        + threadIdx.z * blockDim.y * blockDim.x
                        + threadIdx.y * blockDim.x + threadIdx.x;
}
```

2D grid of 1D blocks

```
__device__ int getGlobalID_2D_1D(){
        int blockId = blockIdx.y * gridDim.x + blockIdx.x;
        int threadId = blockId * blockDim.x + threadIdx.x;
        return threadId;
}
```

2D grid of 2D blocks

```
__device__int getGlobalID_2D_2D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x;
        int threadId = blockId * (blockDim.x * blockDim.y)
                                      + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

2D grid of 3D blocks

```
__device__int getGlobalID_2D_3D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x;
        int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                              + (threadIdx.z * (blockDim.x * blockDim.y))
                              + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

3D grid of 1D blocks

```
__device__int getGlobalID_3D_1D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
        + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * blockDim.x + threadIdx.x;
        return threadId;
}
```

3D grid of 2D blocks

```
__device__int getGlobalID_3D_2D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
                        + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y)
                        + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
}
```

3D grid of 3D blocks

```
__device__int getGlobalID_3D_3D(){
        int blockId = blockIdx.x + blockIdx.y * gridDim.x
                        + gridDim.x * gridDim.y * blockIdx.z;
        int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                        + (threadIdx.z * (blockDim.x * blockDim.y))
                        + (threadIdx.y * blockDim.x) + threadIdx.x;
        return threadId;
```

//Capture the time of execution of kernel

```
#include <cuda.h>
#include <stdlib.h>
#define N 4096 // size of array

int main(int argc, char *argv[])
{
cudaEvent_t start, stop; // using cuda events to measure time
float elapsed_time_ms;

cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 ); // instrument code to measure start time

vectorAdd<<<B,T>>>(dev_a,dev_b,dev_c);
cudaMemcpy(c,dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

cudaEventRecord( stop, 0 ); // instrument code to measure end time
cudaEventSynchronize( stop);
cudaEventElapsedTime( &elapsed_time_ms, start, stop );
printf("Time to calculate results: %f ms.\n", elapsed_time_ms);
}
```

//Display the error message present in CUDA code

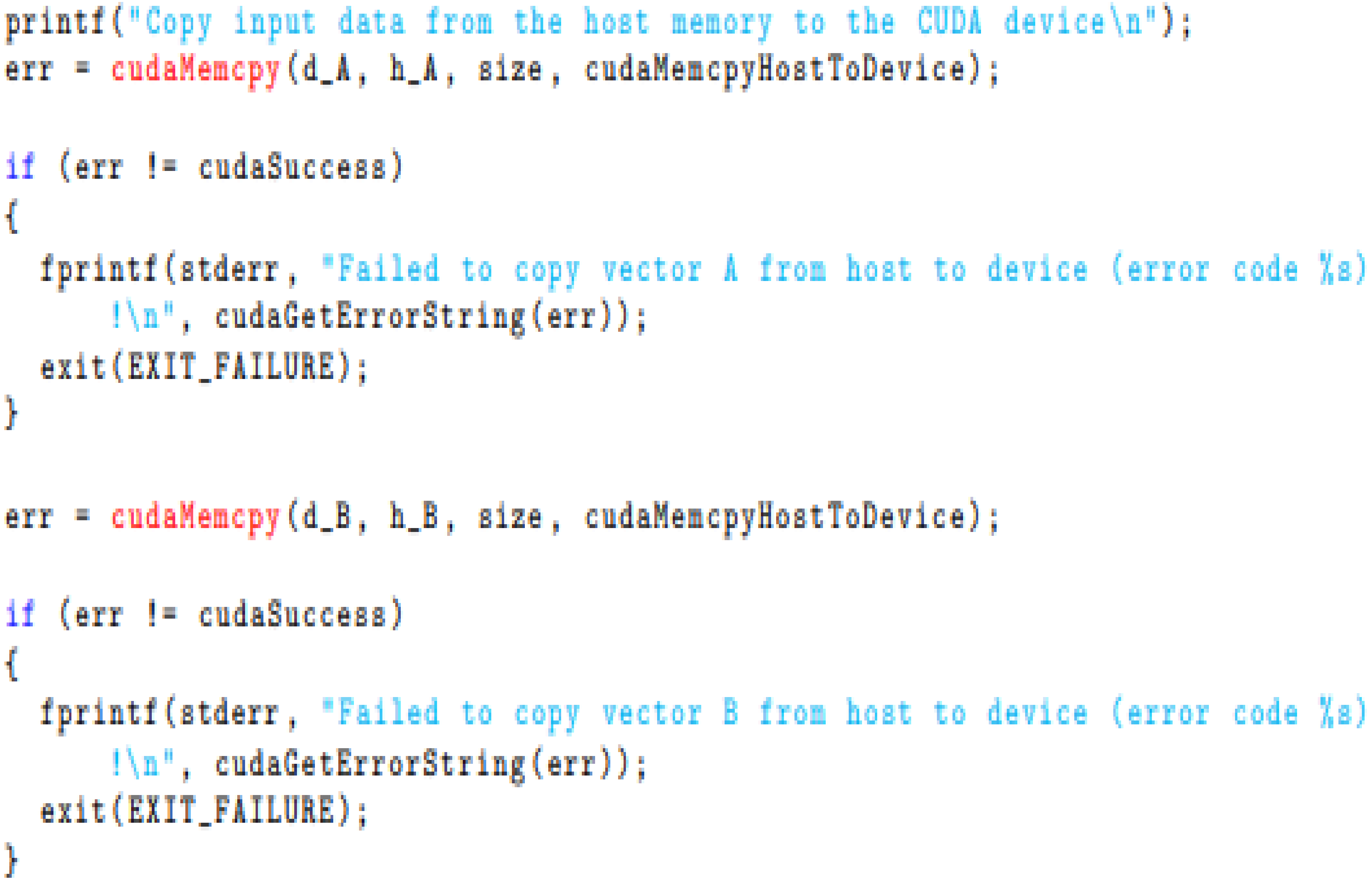
```
cudaError_t cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
   if (cudaStatus != cudaSuccess)
  {
      fprintf(stderr, "cudaMalloc failed!");
      goto Error;
   }

   cudaStatus = cudaMemcpy(pA, A, N*sizeof(char), cudaMemcpyHostToDevice
   if (cudaStatus != cudaSuccess)
  {
      fprintf(stderr, "cudaMmecpy failed!");
      goto Error;
   }
```

A –ve value means there is something wrong in the CUDA code.
It will display the error message present in CUDA code.

Device Memory Allocation

```
err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_B, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Host to Device Data Transfer



```
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector A from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}


err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector B from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

```
//Display the error message present in CUDA code
cudaMemcpy(pA, A, N*sizeof(char), cudaMemcpyHostToDevice);
cudaError_t error =cudaGetLastError();
if (error != cudaSuccess)
{
printf("CUDA Error1: %s\n", cudaGetErrorString(error));
}
CUDAStrCopy<<<1,N>>>(pA,pC);
error =cudaGetLastError();
if (error != cudaSuccess)
{
printf("CUDA Error2: %s\n", cudaGetErrorString(error));
}
```

If call throws an error msg then error is present in CUDA API which precedes this.

If second call throws the error message then error is present in the kernel code.

CUDA kernel invocations do not return any value. Error from a CUDA kernel call can be checked after its execution by calling *cudaGetLastError.*

cudaGetLastError, which reports the last error for any previous runtime call.

cudaGetErrorString uses the error code to display the error string.