CUDA-Introduction

By
Dr. Sandhya Parasnath Dubey and Dr. Manisha
Assistant Professor
Department of Data Science and Computer Applications - MIT,
Manipal Academy of Higher Education, Manipal-576104, Karnataka, India.
Email: sandhya.dubey@manipal.edu; manisha.mit@manipal.edu

Parallelism

- Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand.
- Different variants of parallelism induce different methods of parallelization
- **Instruction Level Parallelism (ILP)**
  - ILP is the parallel or simultaneous execution of a sequence of instructions in a computer program.
  - ILP is specifically about how many instructions a computer can execute at the same time. It measures the efficiency of this parallel execution.
- **Caution:**
  - ➢ Data dependency: Data dependence means that one instruction is dependent on another. This can limit how much parallelism we can achieve.
  - ➢ Name Dependency: A name dependency occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between them. This can also impact parallel execution.

Parallelism

- **Data Level Parallelism**
  - Many problems in scientific computing involve processing of large quantities of data stored on a computer.
  - If we can process different parts of the data simultaneously using multiple processors, this is called **data level parallelism**.
  - This approach is widely used in MIMD (Multiple Instruction, Multiple Data) type computers, where each processor can execute different instruction set (i.e., different parts of the code) on different pieces of data.
  - The same code is executed on all processors, with independent instruction pointers.
  - **Example: If you have a huge dataset that needs to be processed, instead of one processor handling all the data, multiple processors can each work on different sections of the data simultaneously, making the whole process much faster.**

Parallelism

- **Functional/Task Level parallelism**
  - Sometimes the solution of a “big” numerical problem can be split into separate subtasks.
  - A large problem can be broken down into smaller tasks, each of which can be executed independently.
  - Which work together by data exchange and synchronization.
  - In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called **MPMD (Multiple Program Multiple Data)**.
  - Functional parallelism bears pros and cons
    - Cons: When different parts of the problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise. If one task takes longer than another, it can slow down the entire process.
    - Pros: On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably.

Introduction: Benefits of Using GPUs

- The **Graphics Processing Unit (GPU)** provides much higher instruction throughput and memory bandwidth than the CPU
  - For certain types of tasks, the GPU can process a massive amount of data and perform a large number of operations in parallel, far more than a CPU could.
- Many applications leverage these higher capabilities to run faster on the GPU than on the CPU.
- This difference in capabilities between the **GPU** and the **CPU** exists because they are designed with different goals:
  - The CPU is designed to excel at executing a **sequence of operations**, called a **thread**, as fast as possible and can execute a few tens of these threads in parallel. (running your operating system or basic software applications, memory management)
  - The GPU is designed to excel at executing **thousands of threads in parallel.** This makes it extremely efficient at performing the same operation on large datasets—like rendering the pixels on your screen or processing large matrices in deep learning models.
  - The GPU is specialized for highly parallel computations. Its architecture is different from the CPU because more transistors—the basic building blocks of the chip—are dedicated to data processing rather than data caching and flow control.
  - While the CPU is a general-purpose processor that can do a bit of everything, the GPU is a highly specialized tool designed to perform a lot of similar operations very quickly and simultaneously. It is well-suited for tasks that can be broken down into thousands of parallel operations.

Introduction: Benefits of Using GPUs

- Devoting more transistors to data processing.
  - e.g: floating-point computations
- The GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control.
- This approach contrasts with CPUs, which often rely on large data caches and complex flow control mechanisms to minimize the impact of memory access delays.
- In CPU, long memory access latencies are a significant challenge and require expensive solutions like large caches and intricate flow control mechanisms, which use a lot of transistors.
- GPUs, avoid these expensive latencies by focusing on continuous computation. When a thread running on a GPU needs to access data from memory, it might encounter a delay while waiting for that data to be fetched. Instead of stalling the entire GPU while waiting, the GPU can quickly "switch out" that thread and "switch in" another thread that is ready to continue computing, effectively keeping the processing pipeline full and efficient.

Introduction: CPU vs GPU

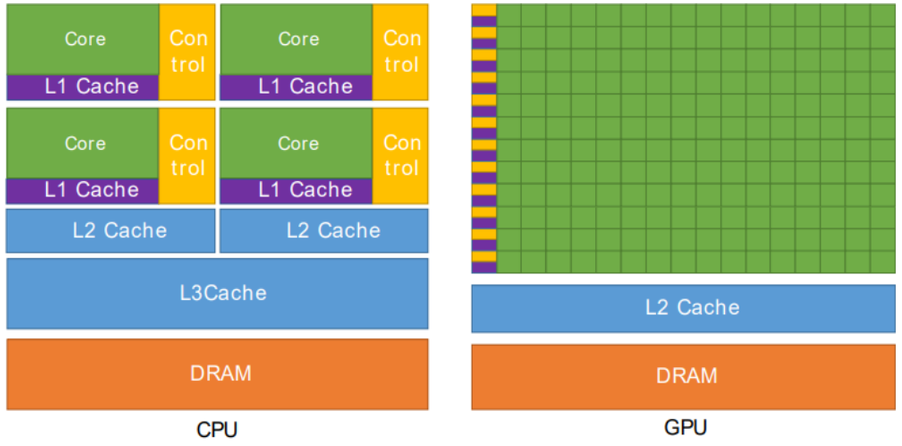


Figure 1. shows an example distribution of chip resources for a CPU versus a GPU

- The CPU has a few large, powerful cores. Each core has its own control unit. These cores are designed to handle a wide range of tasks quickly and efficiently, making CPUs good at general-purpose computing. CPUs have multiple levels of cache (L1, L2, and sometimes L3) that are close to the cores. Below the caches is the DRAM (main memory), which is used to store data and instructions that are not immediately needed but can be fetched into the cache as required.
- The GPU has a large number of smaller, simpler cores arranged in a highly parallel fashion. Each green square in the diagram represents a core. These cores are designed to perform the same operation on multiple pieces of data simultaneously.
- Like the CPU, the GPU also has DRAM for storing data and instructions. However, because the GPU's strength is in handling large amounts of data at once, it often accesses memory in large blocks, which is different from the CPU's approach.

A Scalable Programming Model

- Mainstream processor chips are now built as parallel systems.
  - Multicore CPUs and manycore GPUs
  - **Multicore CPUs:** These have multiple cores (like dual-core, quad-core, octa-core) that can execute multiple threads or tasks at the same time. **Manycore GPUs:** They contain hundreds or even thousands of smaller cores optimized for simultaneous processing.
- The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores
- CUDA (Compute Unified Device Architecture) developed by NVIDIA, is a parallel computing platform and programming model specifically designed for GPUs.
- It provides a framework to write programs that can run on NVIDIA GPUs. It allows developers to write code that can be executed in parallel across many GPU cores.

A Scalable Programming Model

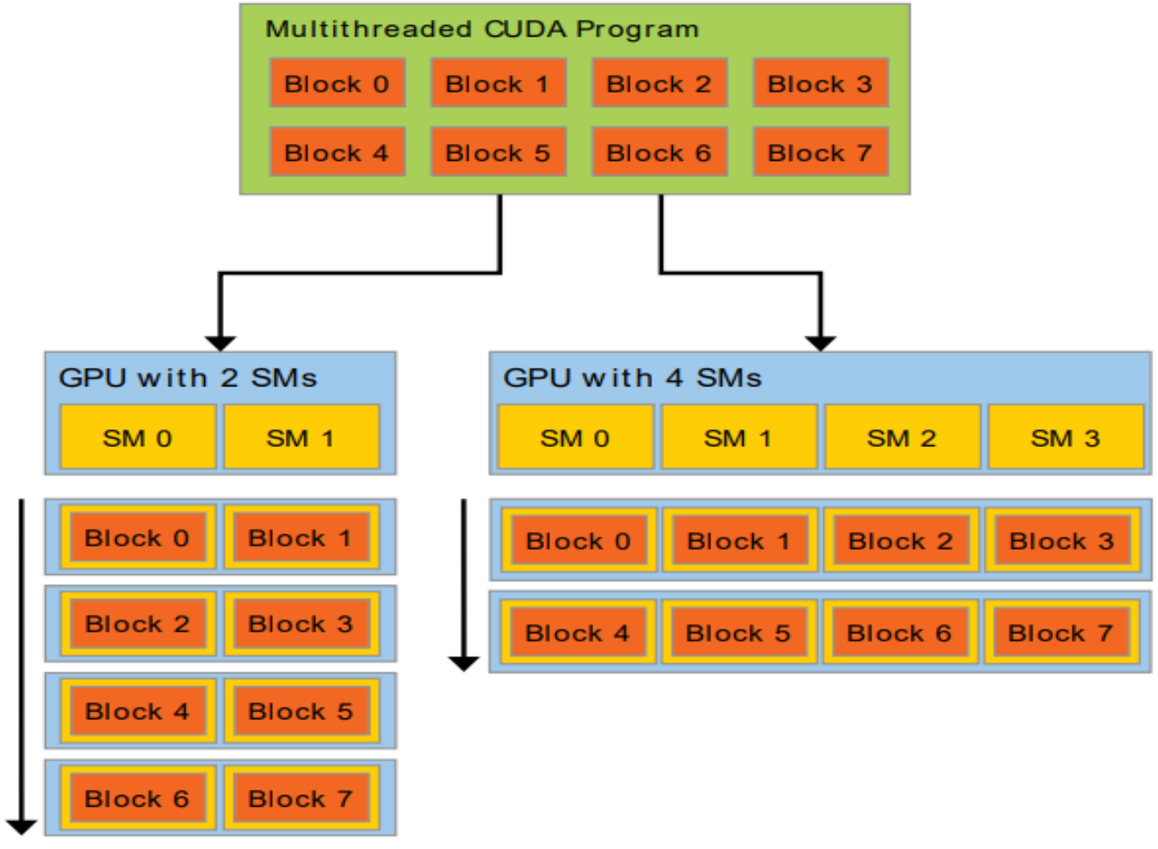
- To help developers achieve this scalable parallelism, CUDA introduces three main concepts:

- **A hierarchy of thread groups:** Each thread executes a copy of the kernel function. A kernel is a function that is executed on the GPU. When you call a kernel, it runs many times simultaneously. Each of these runs is called a "thread," and all these threads work together to complete a task more quickly. CUDA organizes threads into a hierarchy, which allows for scalable parallel execution:
  - **Threads:** The smallest unit of execution.
  - **Thread Blocks:** A group of threads that can cooperate among themselves by sharing data through shared memory and can synchronize their execution to coordinate with each other.
  - **Grid of Thread Blocks:** Multiple thread blocks that can execute independently and in parallel across different cores of the GPU.
- **Shared memories:** CUDA provides different types of memory spaces, including shared memory, which is a fast, on-chip memory that is shared among threads within the same block. It is much faster than global memory (which is off-chip) and can be used for inter-thread communication within a block.
- **Barrier synchronization:** CUDA provides mechanisms to synchronize threads within a block. This ensures that all threads in a block have reached a certain point in the execution before any of them can proceed

A Scalable Programming Model

- CUDA guides the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads.
- Each sub-problem is further divided into smaller tasks that are solved cooperatively by all threads within a block.
- Each block of threads can be scheduled on any of the available multiprocessors within a GPU.
- Blocks may be executed in any order, concurrently (at the same time) or sequentially (one after another), depending on the GPU's resources.
- This flexible scheduling ensures that the same CUDA program can run on GPUs with different numbers of multiprocessors without modification.
- Only the runtime system needs to know the physical multiprocessor count.

A Scalable Programming Model



Blocks in CUDA represent groups of threads that execute the same kernel function. These blocks are independent of each other and can be scheduled on any available SM.

- The image demonstrates CUDA's flexibility in executing the same program on different hardware configurations.
- Whether a GPU has 2 or 4 SMs, the program adapts, and the runtime system manages how the blocks are scheduled across the available SMs.
- **Scalability:** CUDA's programming model is scalable. A program written for one GPU can run on another with more or fewer SMs without modification.
- The CUDA runtime system takes care of distributing the work (blocks) across the available SMs, optimizing performance based on the hardware configuration.

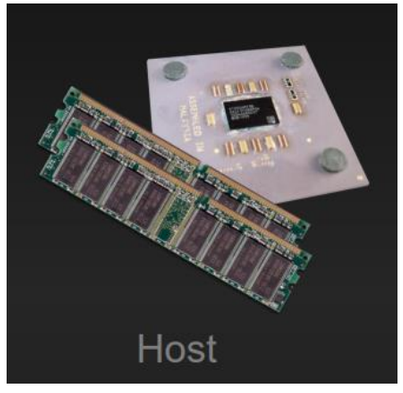
Development environment

Development environment necessary for programming with NVIDIA GPUs using CUDA:

- Every NVIDIA GPU released since the 2006 GeForce 8800 GTX is equipped with CUDA capabilities.
  - GPUs are built on the CUDA architecture, which allows them to run CUDA programs.
- **NVIDIA DEVICE DRIVER**
  - NVIDIA provides specialized system software, known as the device driver, which acts as an intermediary between your programs and the CUDA-enabled hardware.
  - The device driver ensures that your applications can efficiently communicate with and utilize the GPU for computation.
- **CUDA Development Toolkit**
  - CUDA applications typically involve computations on both the GPU and the CPU. You need two compilers:
  - GPU Compiler (CUDA Toolkit): NVIDIA provides a specific compiler as part of the CUDA Toolkit that compiles code to run on the GPU.
  - CPU Compiler: The regular CPU code is compiled using a standard compiler like GCC (on Linux) or MSVC (on Windows).

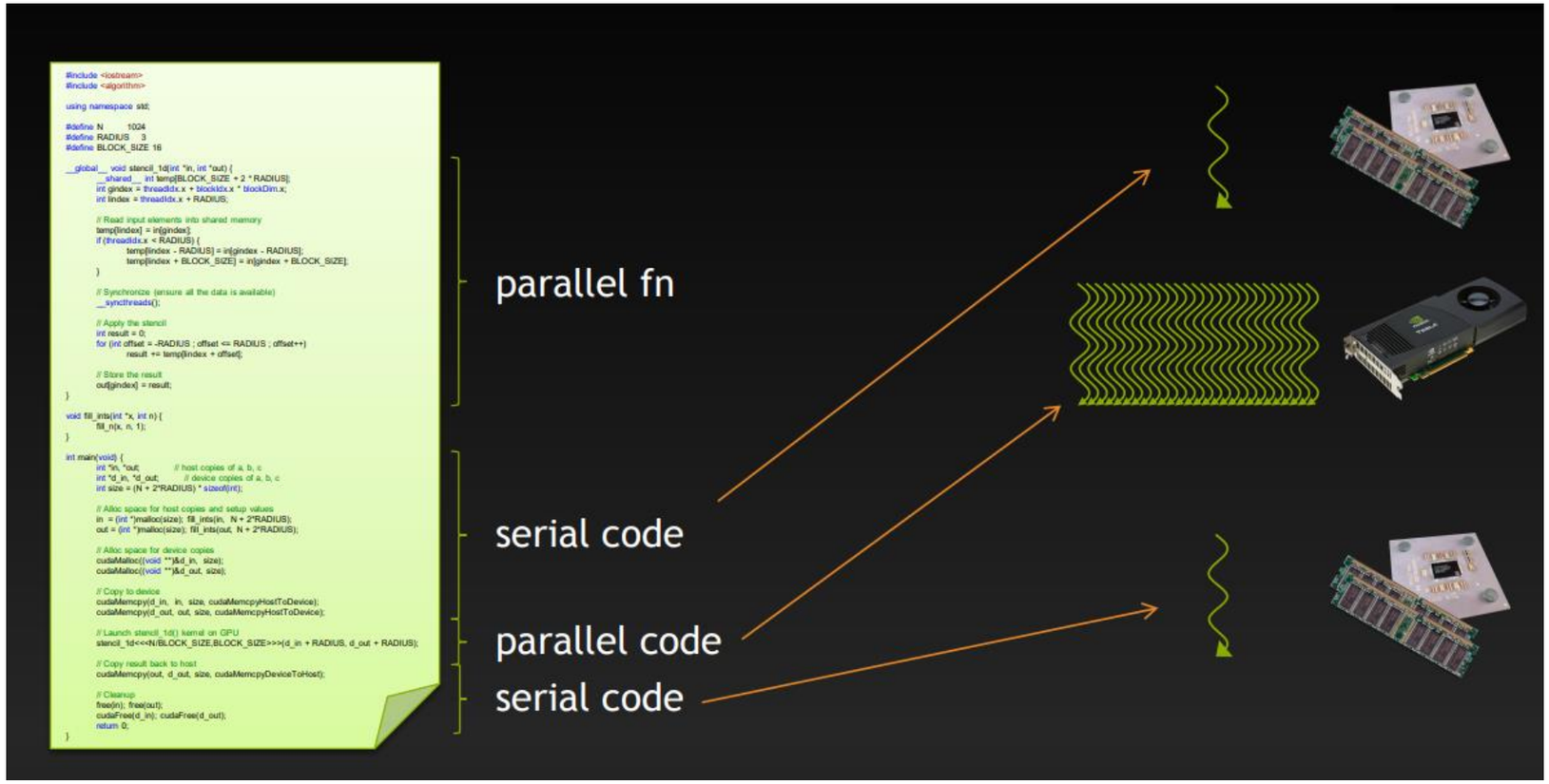
Heterogeneous Computing

- **Heterogeneous computing** refers to the use of multiple types of processors (e.g., CPUs and GPUs) within the same system to perform computational tasks.

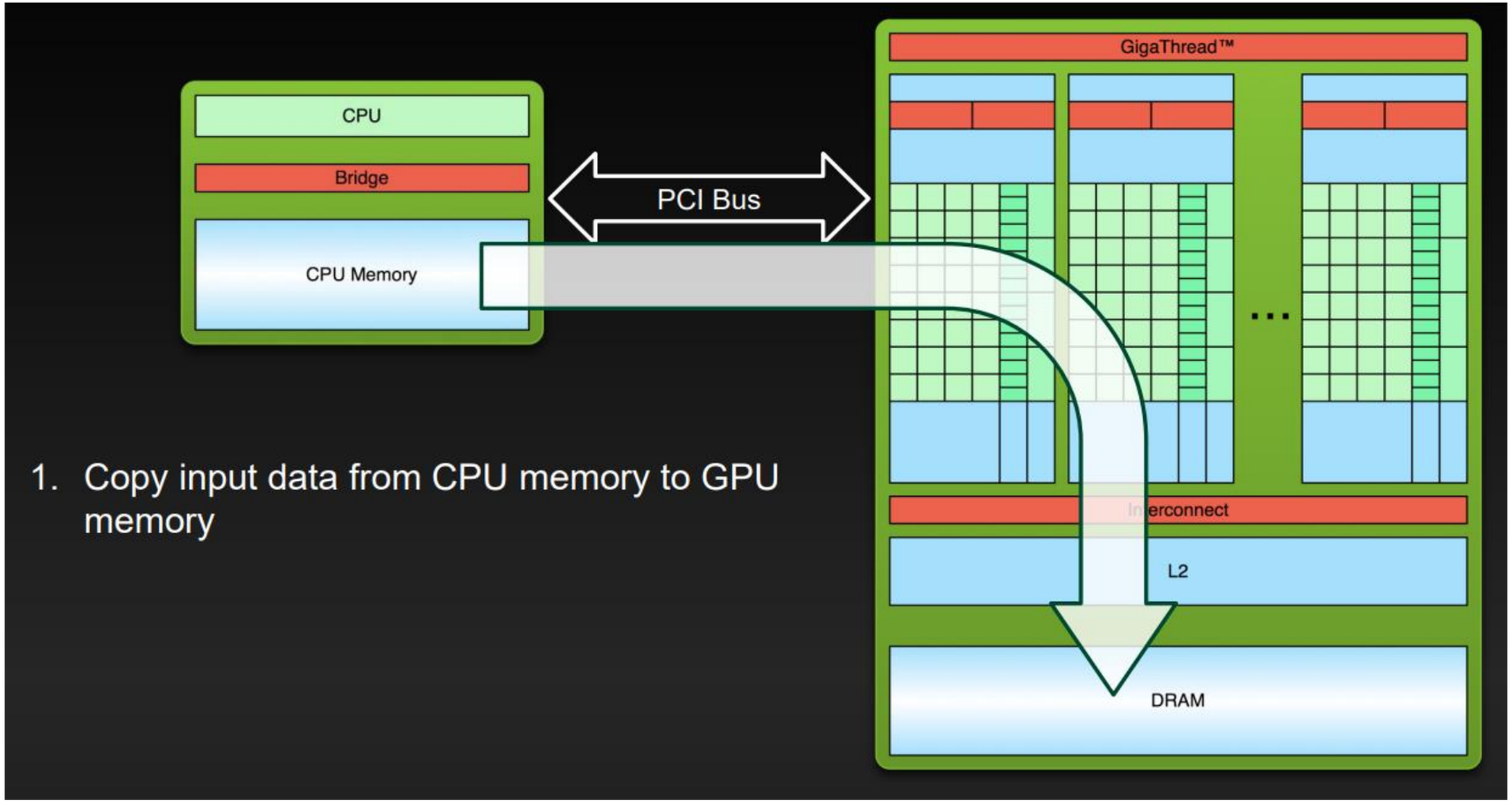


- Terminology:
  - **Host:** The CPU and its memory (host memory)
  - **Host Memory:** This is the system's main memory (RAM) that is accessible by the CPU. The CPU performs general-purpose processing and manages the overall operation of the system.
  - **Role in Heterogeneous Computing:** In CUDA programming, the host is responsible for initiating and managing computational tasks, including preparing data and sending it to the GPU for processing.
  - **Device**: The GPU and its memory (device memory)
  - **Device Memory:** This is the GPU's own memory, separate from the CPU's memory. It is used to store data that the GPU will process and the results it produces.
  - **Role in Heterogeneous Computing:** The device (GPU) handles the parallel processing of data. It executes computationally intensive tasks that have been offloaded by the host.

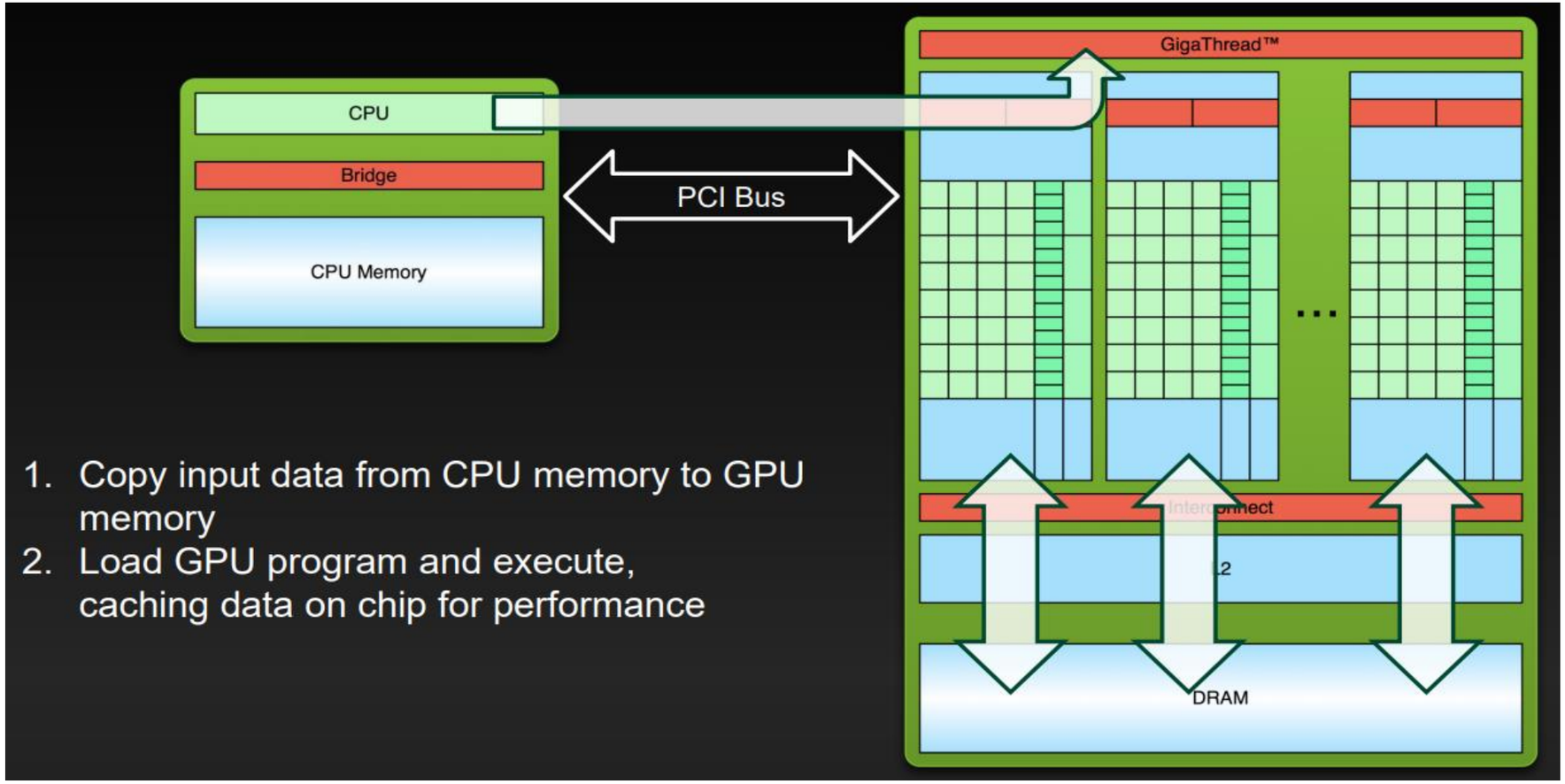
Heterogeneous Computing



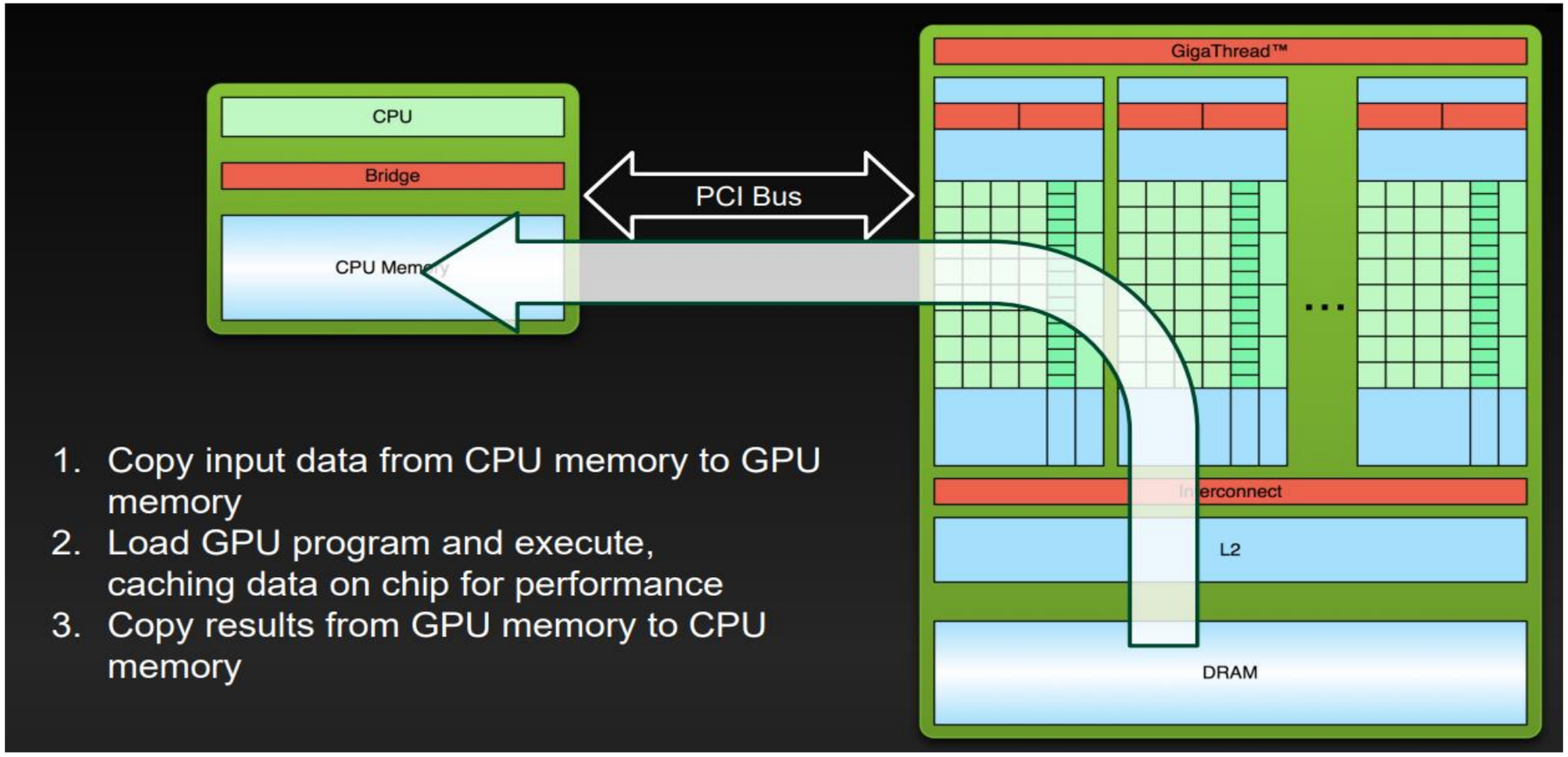
Process flow



Process flow



Process flow



Hello world example

- This is a simple CUDA C program with two distinctions:
  - A function named kernel() qualified with __global__
  - A call to the function, embellished with <<<1,1>>>
- The __global__ qualifier before the function kernel indicates that this function will be compiled to run on the device, i.e., the GPU. The kernel function is the CUDA kernel.
- **Kernel Function:**
  - The __global__ qualifier tells the CUDA compiler to generate device code for this function. When this function runs, it will execute on the GPU.
  - Inside this kernel function, we use printf to print "Hello, CUDA World!" to the GPU's output.
- **Main Function:**
  - The main function is executed on the host, which is the CPU in this context.
  - kernel<<<1, 1>>>(); launches the kernel on the GPU with a grid size of 1 block and 1 thread per block.
  - This is a minimal setup to execute the kernel.
  - cudaDeviceSynchronize(); ensures that the host waits for the GPU to finish executing the kernel before proceeding. This helps in synchronizing the host and device.
- **Compilation and Execution:**
  - When you compile this code using nvcc, the CUDA compiler, it will separate the device code and host code.
  - The kernel() function is compiled by nvcc into code that runs on the GPU.
  - The main() function is compiled by the host compiler, which is typically a standard C compiler.
  - When you run the code, the kernel() function will be executed on the GPU, and you should see "Hello, CUDA World!" printed on the console.

```
#include <stdio.h>

__global__ void kernel() {
    printf("Hello, CUDA World!\n");
}

int main() {
    kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Kernels

- In CUDA, we extend C/C++ with some additional features that allow us to write programs that can run on a GPU.
- CUDA C/C++ extends C/C++ by allowing the programmer to define C/C++ functions, called **kernels**.
- When called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C/C++ functions.
- A kernel is defined using the **__global__** declaration specifier.
- The number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<< >>>** execution configuration syntax.
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.
- This allows each thread to work on a different piece of data. In this case, each thread will add one element from the arrays a and b and store the result in c.

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x; // Each thread has a unique thread ID
    c[index] = a[index] + b[index]; // Each thread computes one element of c
}
```

Kernels

- In this line of code, <<<1, N>>> is the execution configuration.
- It tells CUDA how many threads to launch.
- The first number, 1, is the number of thread blocks, and the second number, N, is the number of threads per block.
- In the example, we are launching N threads, all executing the same kernel function add in parallel. Each thread operates on different data based on its thread ID.
- This ID is crucial because it allows each thread to work on a different piece of data. In the add kernel, each thread uses its ID to determine which elements of the arrays a and b it should add together.

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x; // Each thread has a unique thread ID
    c[index] = a[index] + b[index]; // Each thread computes one element of c
}
```

```
int N = 256;
add<<<1, N>>>(a, b, c);
```

Kernels

- In this example, VecAdd<<<2, N>>> launches a kernel with a grid of 2 blocks and each block containing N threads.
- Each thread computes one element of the vector addition, effectively processing 2 * N elements.
- The total number of threads in the grid is 2 * N. Each thread is assigned a unique index that can be calculated using the block and thread indices. This index helps each thread process a different element or piece of data.
- Here, blockIdx.x gives the block index, blockDim.x gives the number of threads per block, and threadIdx.x gives the thread index within its block.
- Each block can be executed concurrently on different Streaming Multiprocessors (SMs) on the GPU. If there are enough SMs available, both blocks can be processed in parallel, improving overall performance.

```
__global__ void VecAdd(float *A, float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // Compute the global index

    if (i < N) {
        C[i] = A[i] + B[i];  // Perform the vector addition
    }
}
```

```
int N = 1024;  // Size of the vectors
float *d_A, *d_B, *d_C;

// Allocate memory on device
cudaMalloc(&d_A, N * sizeof(float));
cudaMalloc(&d_B, N * sizeof(float));
cudaMalloc(&d_C, N * sizeof(float));

// Launch the kernel
VecAdd<<<2, N>>>(d_A, d_B, d_C, N);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Kernels

- The following sample code, using the built-in variable threadIdx, adds two vectors A and B of size N and stores the result into vector C.
- Each of the N threads that execute VecAdd() performs one pair-wise addition.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

```cpp
#include <stdio.h>
#include <cuda_runtime.h>
#include <stdlib.h>

__global__ void VecAdd(float* A, float* B, float* C, int N) {
    int i = threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int N = 512;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    for (int i = 0; i < N; ++i) {
        h_A[i] = i;
        h_B[i] = i * 2;
    }

    // Allocate vectors in device memory
    float* d_A;
    float* d_B;
    float* d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    VecAdd<<<1, N>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Verify result
    for (int i = 0; i < N; ++i) {
        if (h_C[i] != h_A[i] + h_B[i]) {
            printf("Error at index %d\n", i);
            break;
        }
    }

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    printf("Completed successfully!\n");
    return 0;
}
```

Thread Hierarchy

- **threadIdx** is a built-in variable in CUDA that gives each thread a unique identifier within a thread block.
- It's a 3-component vector **(threadIdx.x, threadIdx.y, threadIdx.z).**
- It can represent a thread's position in a one-dimensional (1D), two-dimensional (2D), or three-dimensional (3D) thread block.
- Thread Blocks are groups of threads that execute on a GPU core.
- A thread block can be organized in 1D, 2D, or 3D.
- For example, if you have a block of size $\boldsymbol{Dx}$, then the threads are organized in a 1D structure.
- For a block of size $\boldsymbol{(Dx, Dy)}$, threads are organized in a 2D structure.
- For $\boldsymbol{(Dx, Dy, Dz)}$, threads are organized in 3D.
- The index of a thread and its thread ID relate to each other in a straightforward way.

Thread Hierarchy

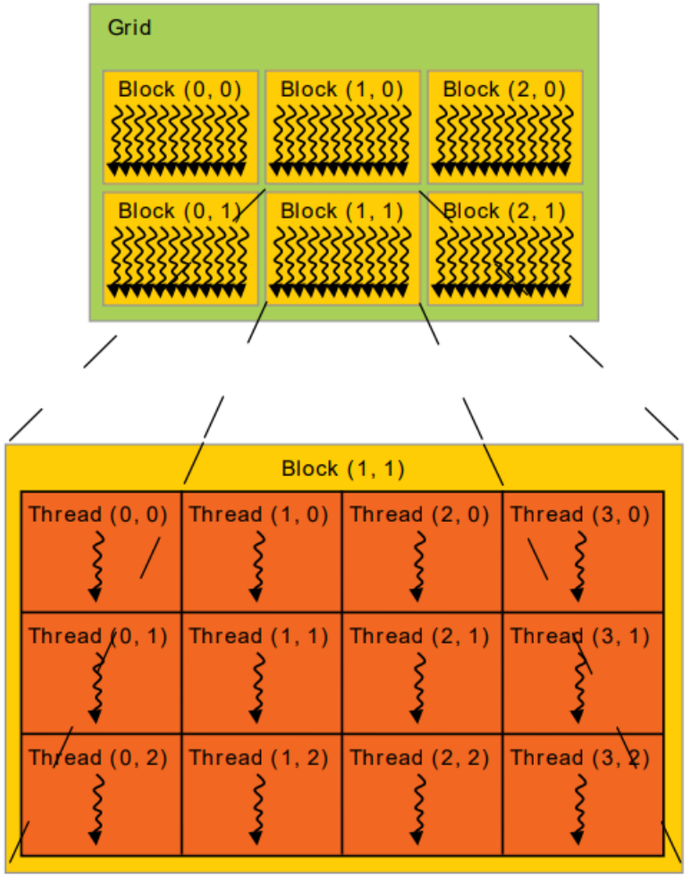
- The relationship between a thread's index and its thread ID varies depending on the dimensionality of the block:
- 1D Block: The thread ID is directly given by **threadIdx.x**.
- 2D Block: The thread ID is calculated as **threadIdx.x + threadIdx.y * blockDim.x**.
- 3D Block: The thread ID is calculated as **threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y.**
- This allows each thread to have a unique ID within a block, which helps in dividing tasks among threads.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Kernel Functions and Threading

- The number of threads per block and the number of blocks in a grid are specified using the special **<<< >>>** syntax when you launch a kernel.
- For example, **<<<gridSize, blockSize>>>** where **gridSize** defines the number of blocks and **blockSize** defines the number of threads per block.
- Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in **blockIdx** variable.
- **blockIdx**: Identifies the block within the grid.
- **blockDim**: Specifies the dimensions of the thread block.



Kernel Functions and Threading

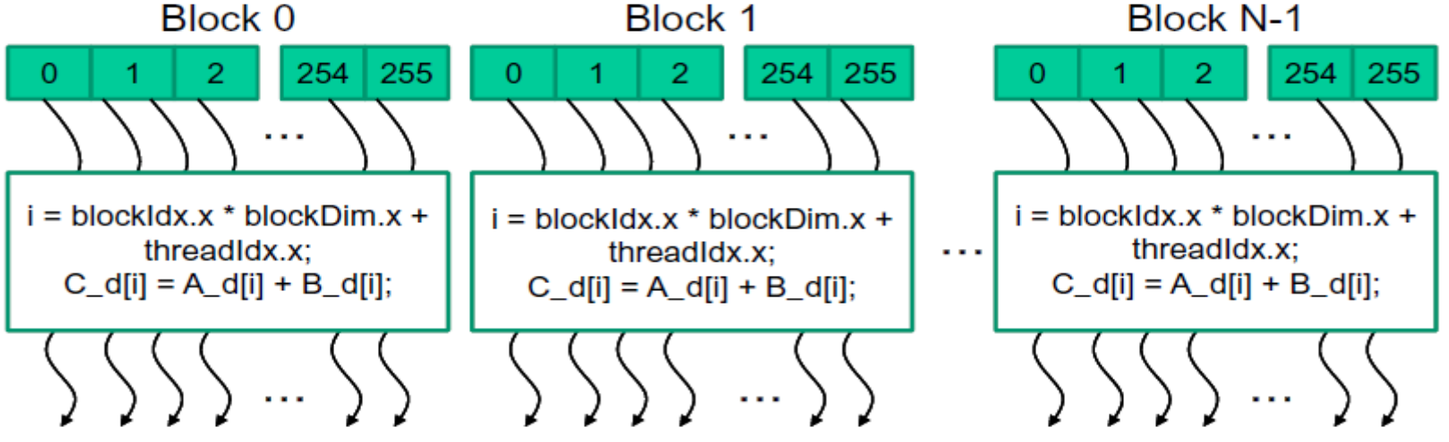
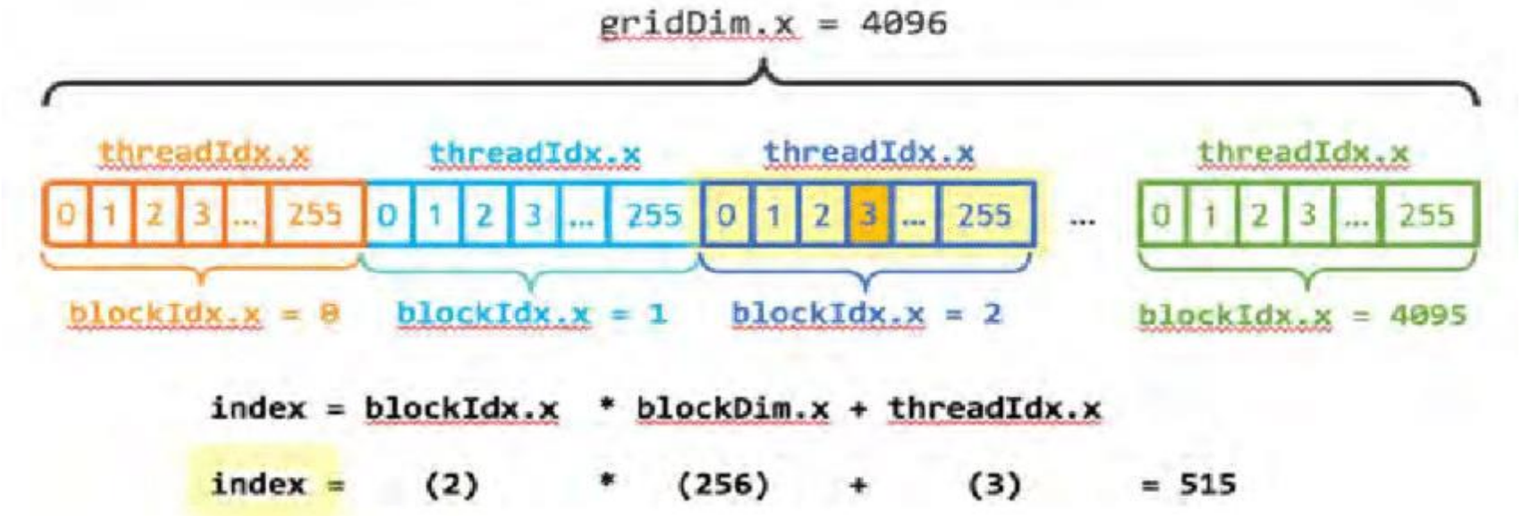


FIGURE 3.10

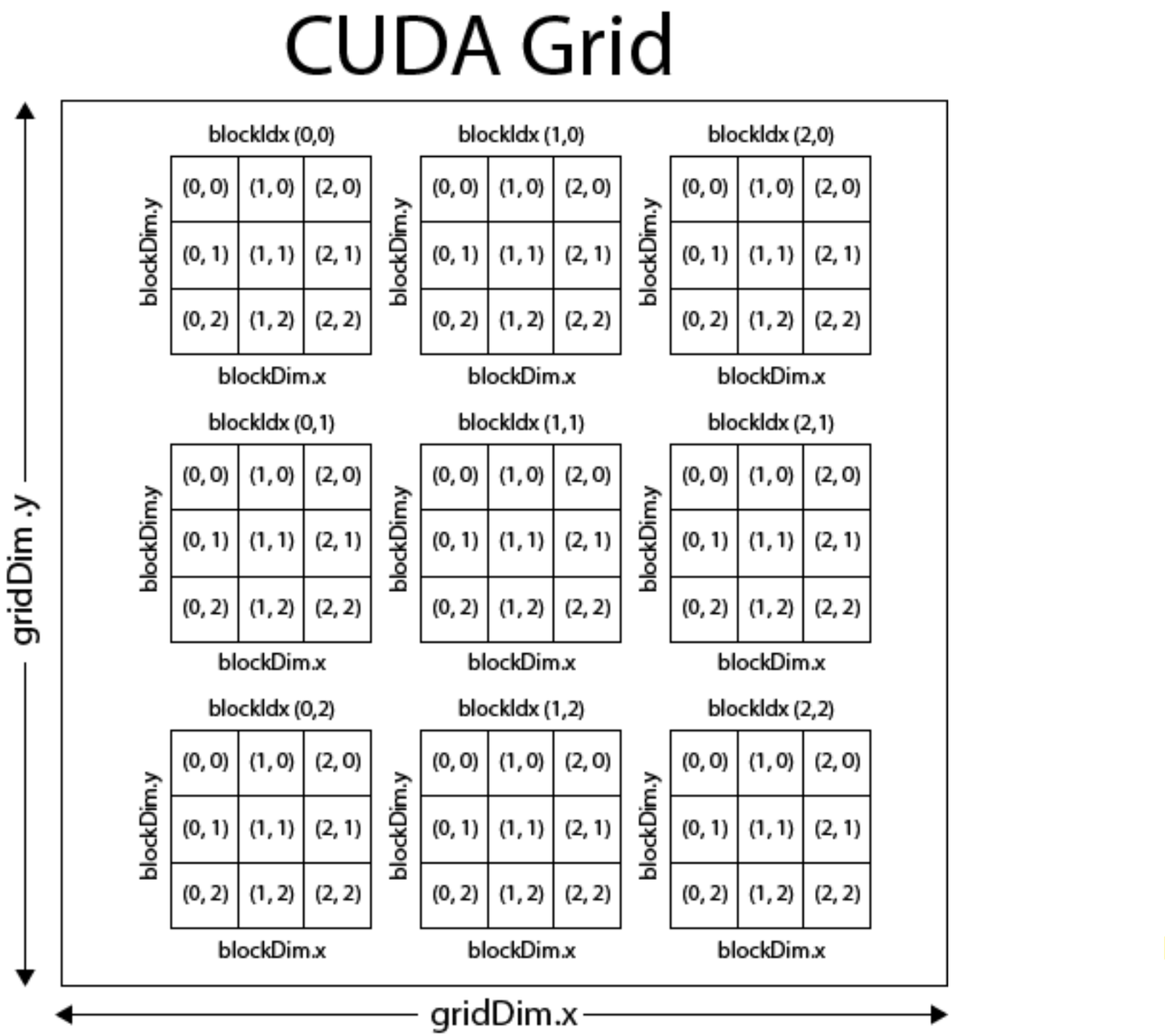
All threads in a grid execute the same kernel code.

- When a host code launches a kernel, the CUDA runtime system generates a grid of threads.
- Grid is organized into an array of thread blocks. All blocks of a grid are of same size.
- Each block can contain upto 1024 threads, with flexibility in distributing these elements into three dimensions.
- The number of threads in each 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 general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons.

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.
- Data index i is calculated as i = blockIdx.x * blockDim.x + threadIdx.x
- Since blockDim is 256 in the example, the i values of threads in block 0 ranges from 0 to 255, in block 1 ranges from 256 to 511, and so on.


CUDA Grid
gridDim .y
gridDim.x
blockIdx (0,0)
blockIdx (1,0)
blockIdx (2,0)
blockIdx (0,1)
blockIdx (1,1)
blockIdx (2,1)
blockIdx (0,2)
blockIdx (1,2)
blockIdx (2,2)
blockDim.y
blockDim.x
(0, 0) (1, 0) (2, 0)
(0, 1) (1, 1) (2, 1)
(0, 2) (1, 2) (2, 2)

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.

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

CUDA Thread Organization

- Execution configuration parameters in a kernel launch statement specify the dimension of the grid and the dimensions of each block.
- In general, 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 organization of a grid is determined by the execution configuration parameters of the kernal launch statement.
- The first parameter specifies the dimensions of the grid in number of blocks.
- The second parameter specifies the dimensions of each block in number of threads.
- Each such parameter is of dim3 type, which is a C struct with three unsigned integer fields, x, y, z.
- For 1D and 2D grids and blocks, the unused dimension fields should be set to 1.
- Example: Generate a 1D grid that consists of 128 blocks, each of which consists of 32 threads. Total number of threads in the grid is 128x32 = 4096. dimBlock and dimGrid are host code variables defined by the programmer.

```
dim3 dimBlock(32, 1, 1);  // 32 threads per block (1D)
dim3 dimGrid(128, 1, 1);  // 128 blocks (1D)
vecAddKernel<<<dimGrid, dimBlock>>>(...);  // Launch the kernel
```

- In CUDA C, the allowed values for gridDim.x, gridDim.y, and gridDim.z ranges from 1 to 65,536.

CUDA Thread Organization

- The kernel launch can also be written as:

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

- It 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 1000, the grid will consist of 4 blocks. The statement will launch 4*256 = 1024 threads. The first 1000 threads will perform addition on the 1000 vector elements and remaining 24 will not.
- If n is equal to 4000, the grid will have 16 blocks. In each case, there will be enough threads to cover all the vector elements.
- Once vecAddKernel() is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.
- CUDA C provides a special shortcut for launching 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>>>(...);
```

- CUDA C compiler simply takes the arithmetic expression as the x dimensions and assumes that the y and z dimensions are 1.

CUDA Thread Organization

- In CUDA, a thread block is a group of threads that execute together on a single multiprocessor.
- In the provided example, we define a thread block of size 16×16, meaning each block contains 256 threads.
- A grid is a collection of thread blocks. In the example, we create a grid that has enough blocks to cover the entire matrix.
- The grid is sized so that each element in the matrices A, B, and C is handled by a separate thread.
- The kernel MatAdd is defined to perform matrix addition. Each thread in this kernel is responsible for computing one element of the output matrix C.
- The kernel uses threadIdx and blockIdx to determine the indices i and j of the matrix element that a particular thread is responsible for.
- threadIdx.x and threadIdx.y give the thread's position within its block, while blockIdx.x and blockIdx.y give the block's position within the grid.
- These are combined with blockDim.x and blockDim.y, which provide the dimensions of the block, to calculate the exact position of the matrix element each thread will process.
- The kernel is launched with a grid and block configuration using the <<<>>> syntax. In this case, the grid is configured to have enough blocks such that each thread processes a unique element in the matrix.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

CUDA Thread Organization

- One of the most important concepts in CUDA is that thread blocks must execute independently. This means that any block can be scheduled on any multiprocessor at any time, without any dependencies on other blocks.
- This independence allows CUDA programs to scale with the number of cores on a GPU.
- As the number of cores increases, more blocks can be executed simultaneously, leading to better performance.
- Although blocks are independent of each other, threads within a block can cooperate. They can share data through shared memory, a special type of memory that is accessible only to threads within the same block.
- Threads within a block can also synchronize their execution using the __syncthreads() function.
- This is important when threads need to wait for each other to reach a certain point before proceeding, for example, to avoid race conditions when accessing shared memory.
- By organizing threads into blocks and grids, CUDA allows the same code to run efficiently on GPUs with different numbers of cores.
- The independence of thread blocks ensures that as more cores become available, the GPU can automatically distribute work among them, leading to improved performance without requiring changes to the code.

CUDA Thread Organization

- The grid can have higher dimensionality than its blocks and vice versa.
- Figure shows an example of 2D grid of size (2, 2, 1), that consists of 3D (4, 2, 2) blocks.
- The grid can be generated with the following host code:

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

- The grid consists of 4 blocks organized into a 2X2 array.
- Each block is labeled with (blockIdx.y, blockIdx.x).
- Example, block(1,0) has blockIdx.y=1 and blockIdx.x=0.
- The ordering of the labels is such that the highest dimensions comes first.
- This is reverse of the ordering used in the configuration parameters where the lowest dimensions comes first.

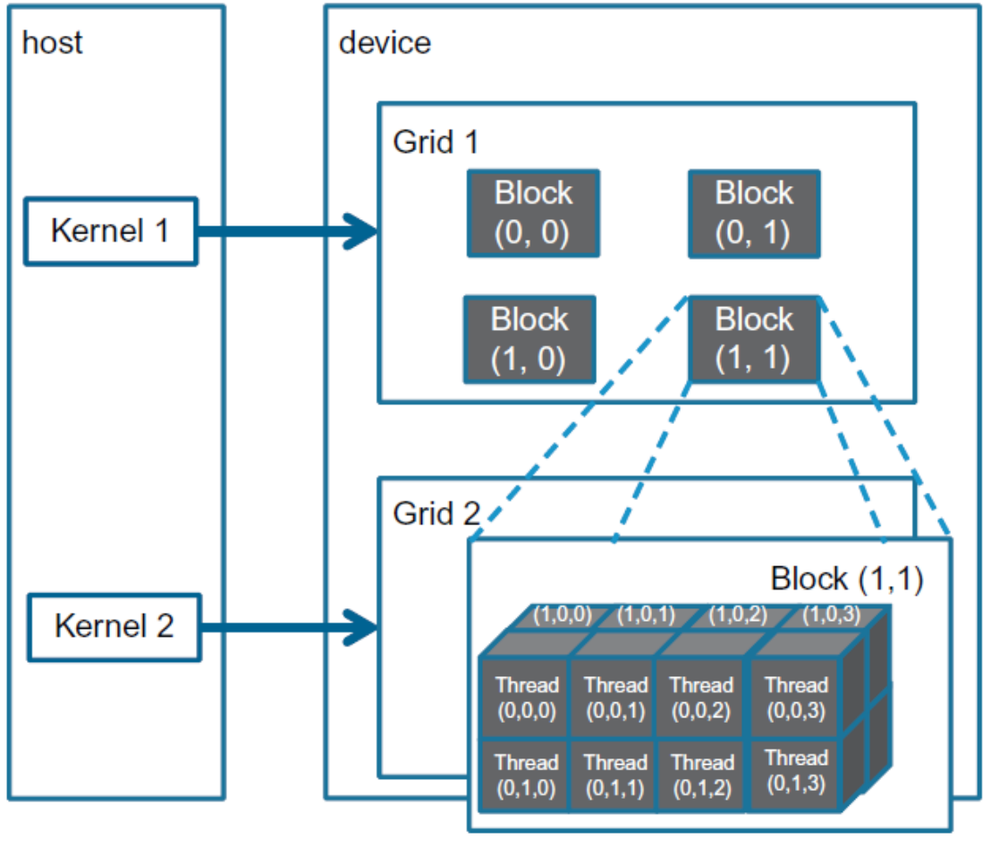


FIGURE 4.1

A multidimensional example of CUDA grid organization.

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.
- Consider a picture of 76 x 62 picture.
- Assume that we decided to use a 16 x 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 x 4 = 20 blocks.
- Note: We have 4 extra threads in the x direction and 2 extra threads in the y direction. That is, we will generate 80x64 threads to process 76x62 pixels.
- An if statement is needed to prevent the extra threads from taking effect. Analogously, we should expect that the picture processing kernel function will have if statements to test whether the thread indices threadIdx.x and threadIdx.y fall within the valid range of pixels.

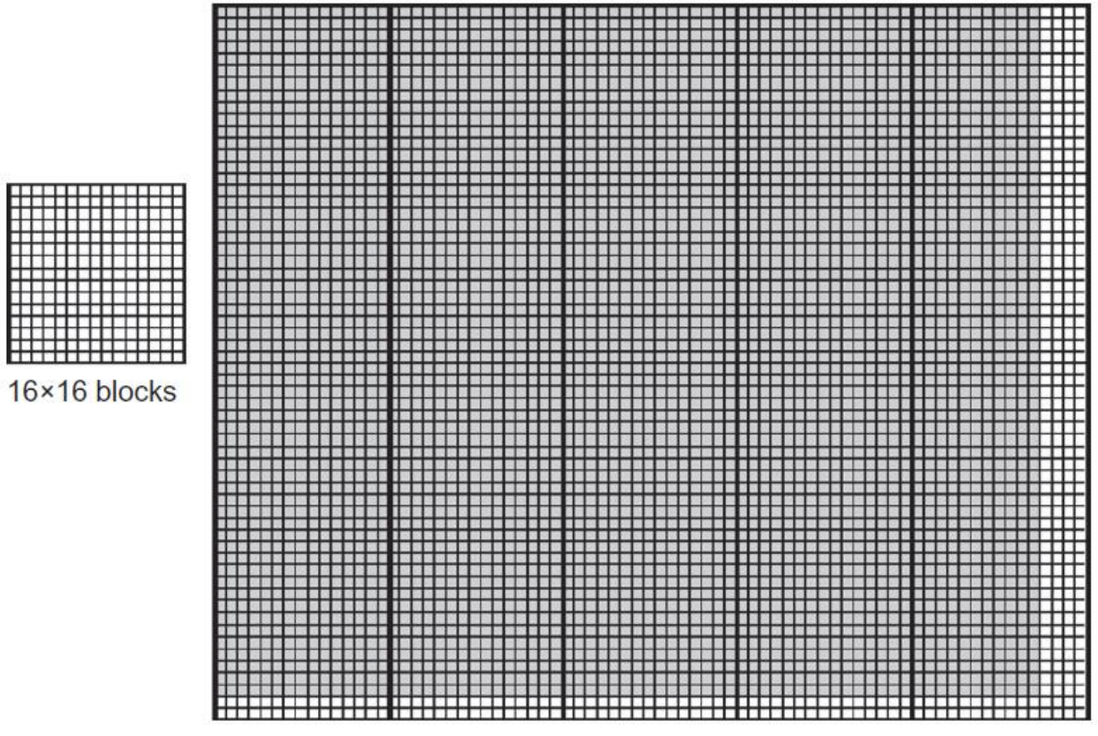


FIGURE 4.2

Using a 2D grid to process a picture.

Mapping Threads to Multidimensional Data

- Assume that the host code uses an integer variable n to track the number of pixels in the x direction, and another integer variable m to track the number of pixels in the y direction.
- We further assume that the input picture data has been copied to the device memory and can be accessed through a pointer variable d_Pin.
- The output picture has been allocated in the device memory and can be accessed through a pointer variable d_Pout.
- The following host code can be used to launch a 2D kernel to process the picture.

```
dim3 dimBlock(16, 16, 1);                                    // Block size: 16x16 threads
dim3 dimGrid(ceil(n / 16.0), ceil(m / 16.0), 1);             // Grid size: based on n and m
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m); // Kernel launch
```

Mapping Threads to Multidimensional Data

```
// 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.

```
__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];
  }

}
```

Mapping Threads to Multidimensional Data

- Ideally, we would like to access d_Pin as a 2D array where an element at row j and column i can be accessed as d_Pin[j][i].
- Programmers need to explicitly linearize, of flatten a dynamically allocated 2D array into 1D array in CUDA C.
- In reality, all multidimensional arrays in C are linearized due to the use of flat memory space in modern computers.
- There are two ways to linearize a 2D array.
- **Row-major layout:** Place all elements of the same row into consecutive locations. It is used by CUDA C
- Rows are then placed one after another into the memory space.
- For 4x4 matrix M, the 1D equivalent index for the M element in row j and column i is **j x 4 + i.**
- The **j x 4** term skips all elements of the rows before row j and **i** term then selects the right element within the section for row j.
- For example: 1D index for M(2,1) is 2x4+1 = 9.
- M_9 is the 1D equivalent to $M_{2,1}$
- **Column-major layout:** Place all elements of the same column into consecutive locations. The columns are then placed one after another into memory space. It is used by FORTRAN compilers.

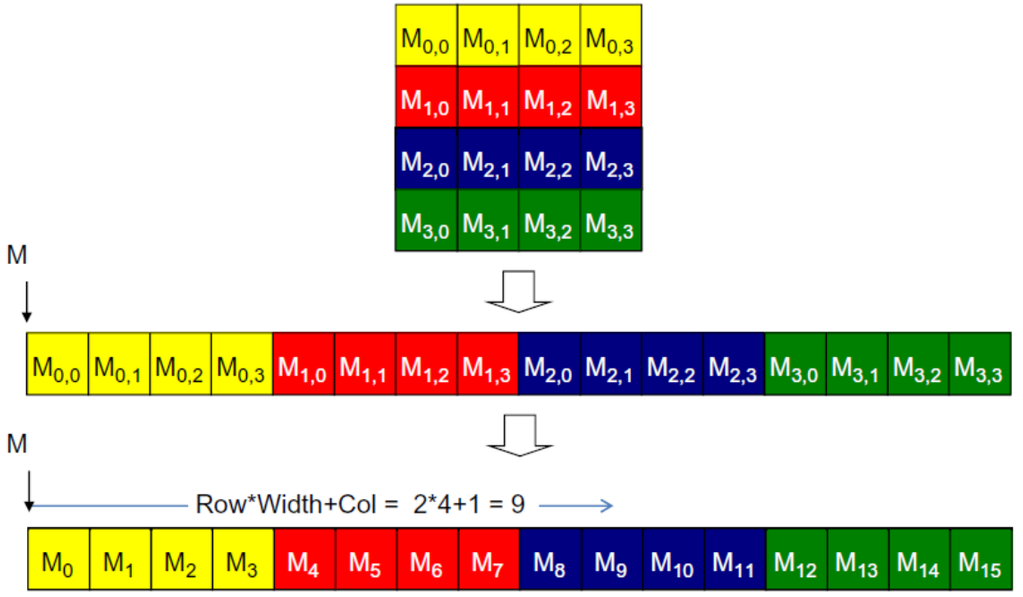


FIGURE 4.3

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.

Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution.

1. Per-Thread Local Memory:

- Each thread has access to its own private **local memory**. This memory is very small and is used to store variables that are private to that thread.
- If a thread is working on a particular element of an array, it can store temporary variables here. This memory is relatively slow if used extensively because it spills over to global memory when registers are full.

2. Per-Block Shared Memory:

- Each block of threads has access to **shared memory**, which is extremely fast and is local to the block. All threads within a block can share data through this memory.
- If multiple threads need to work on a common dataset, shared memory is ideal because it's much faster than accessing global memory repeatedly.
- Reduces access to global memory by storing frequently used data locally within the block.

3. Grid and Global Memory:

- Each grid consists of multiple blocks. All threads from all blocks can access the global memory, which is the largest but slowest form of memory in the CUDA architecture.
- This is where the main data resides. All threads, irrespective of which block they belong to, can access this memory.
- When you copy data from the host (CPU) to the device (GPU), it gets stored in global memory.

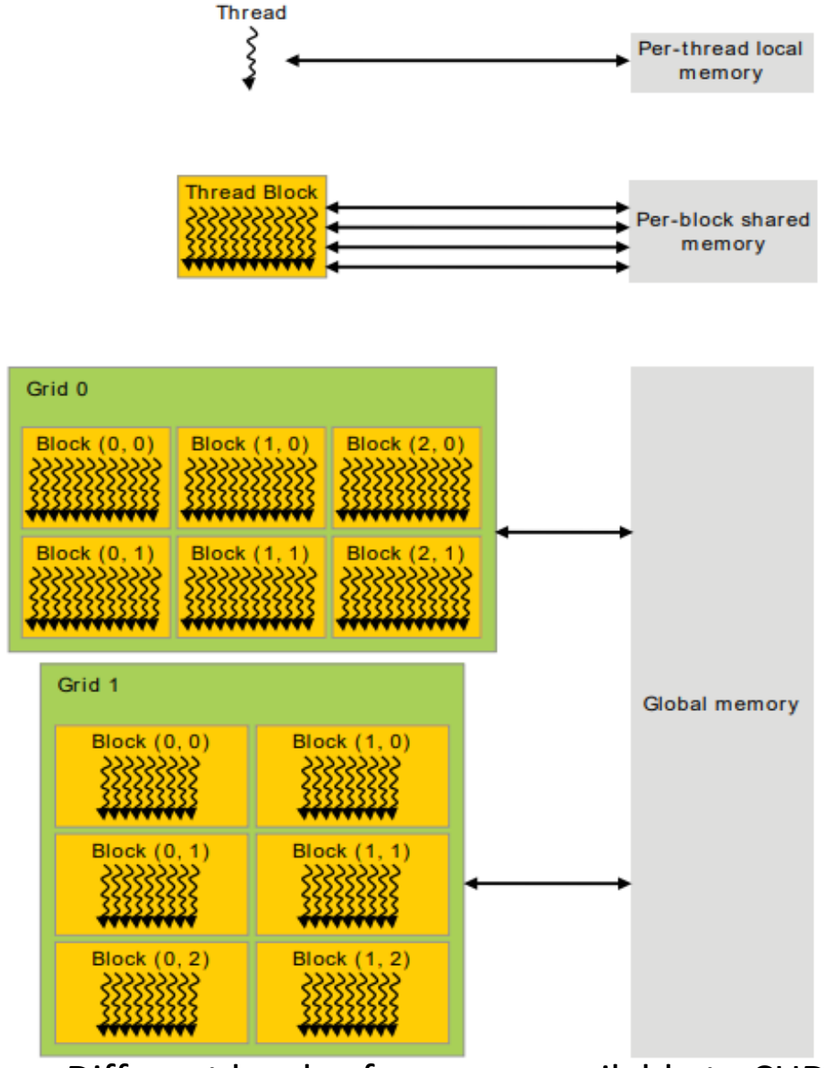


Figure: Different levels of memory available to CUDA threads and how they interact.

Memory Hierarchy

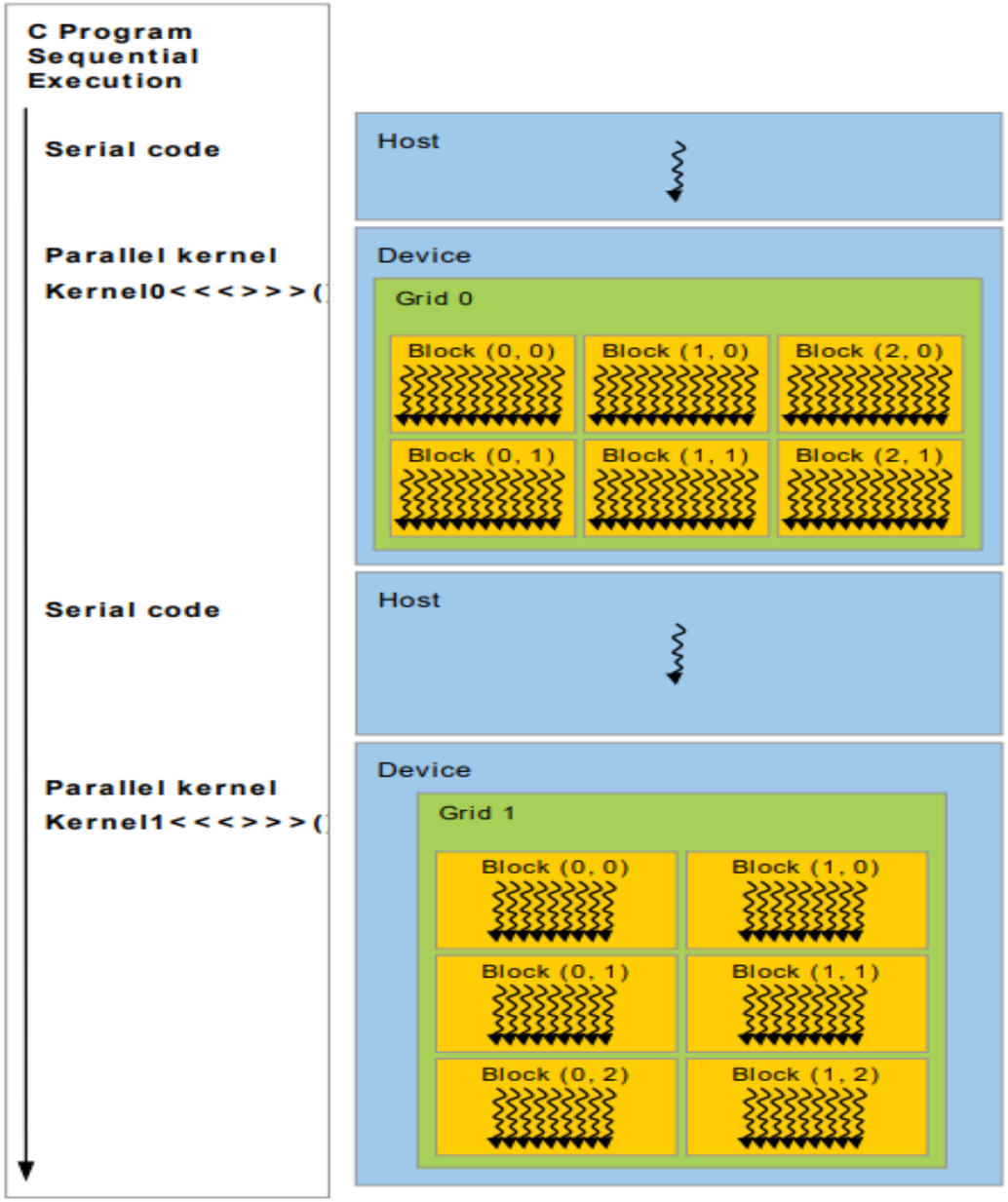
4. Constant and Texture Memory:

- In addition to shared and global memory, CUDA also offers read-only **constant** and **texture** memory.
- **Constant Memory:** This is used when many threads need read-only access to the same data (e.g., a constant value used in all threads).
- It is optimized for scenarios where many threads need to read the same data but the data itself doesn't change during kernel execution. This is ideal for broadcasting the same constant values across multiple threads.
- Threads can read from constant memory, but they cannot modify it.
- **Texture Memory:** This is optimized for 2D spatial locality (i.e., adjacent threads read adjacent memory locations). This is particularly useful certain graphics and image processing applications.
- It is highly effective when threads access data that is stored in nearby memory locations (like neighboring pixels in an image).
- Constant and texture memory resides in global memory on the GPU, but it has a dedicated cache, making access faster than normal global memory.

Heterogeneous Programming

- Heterogeneous computing refers to the use of multiple types of processors (e.g., CPUs and GPUs) within the same system to perform computational tasks.
- CUDA assumes that the CPU (host) and GPU (device) are physically separate entities.
- The CPU executes the main program, while the GPU runs the parallel code (kernels) designed to be executed by thousands of threads simultaneously.
- The GPU acts as a coprocessor to the CPU. It performs computationally intensive tasks while the CPU handles the overall program logic and coordination. The CPU launches kernels that run on the GPU.
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory.
- Therefore, data needs to be transferred between them.
- CUDA provides a set of API calls to manage this process.
- This includes device memory allocation and deallocation as well as data transfer between host and device memory.

Heterogeneous Programming



Streaming Multiprocessors

Threads and Warps:

- In CUDA, threads are grouped into **warps**.
- A warp is a collection of 32 threads that execute the same instructions simultaneously (SIMT—Single Instruction, Multiple Threads). Multiple warps together form a thread block.

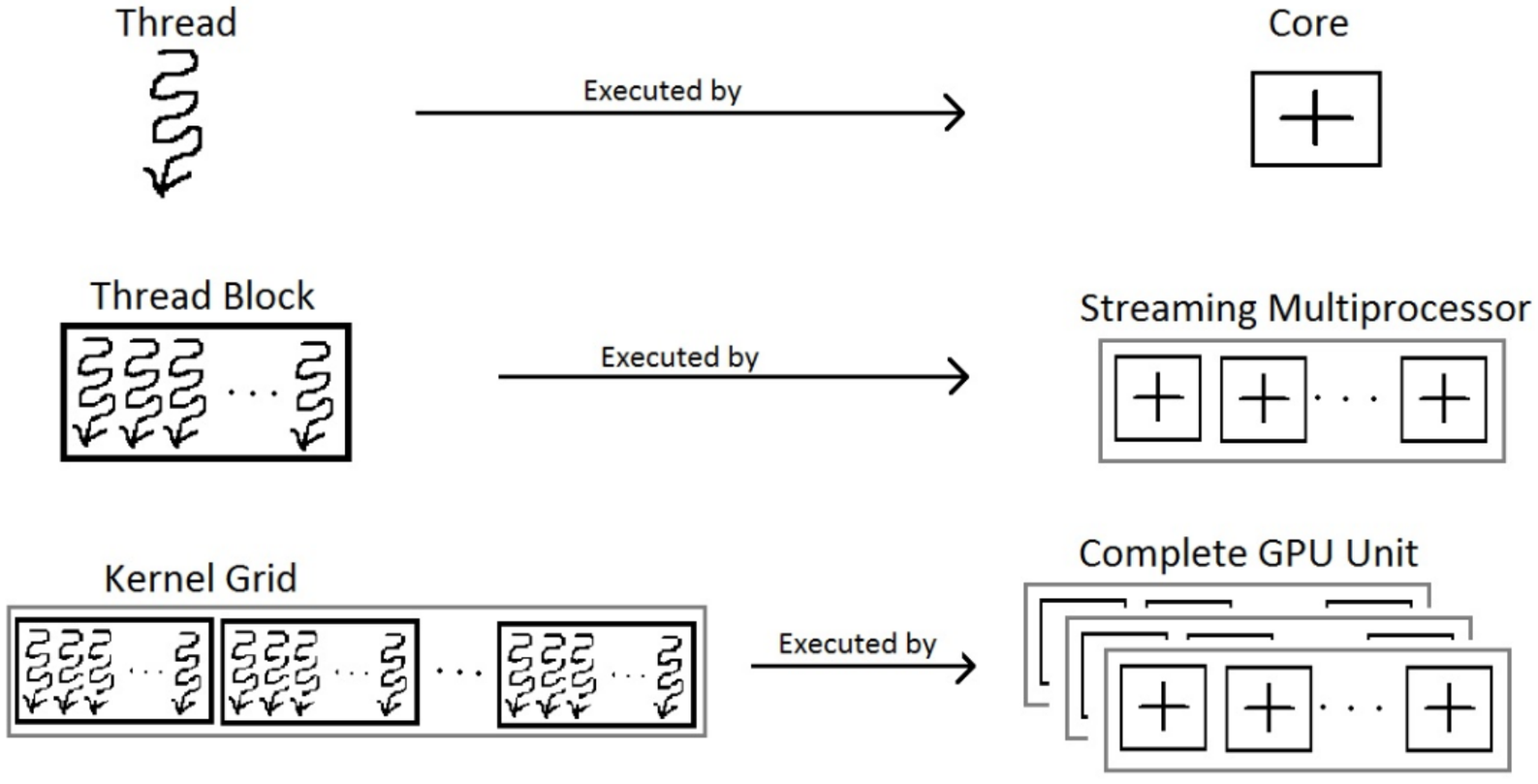
Thread Blocks and SMs:

- Several thread blocks are assigned to a Streaming Multiprocessor (SM) for execution.
- Each SM is responsible for executing multiple thread blocks in parallel.
- Once one thread block finishes, the SM immediately begins executing the next thread block in the queue.

Multiple SMs in a GPU:

- A GPU is composed of multiple SMs, each of which can run several thread blocks at the same time.
- This enables massive parallelism in the GPU.
- 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.** Each SM contains multiple execution cores capable of handling different types of instructions:
    1. Single Precision Floating Point Units: For operations using single-precision (32-bit) floating-point numbers.
    2. Double Precision Floating Point Units: For operations using double-precision (64-bit) floating-point numbers.
    3. Special Function Units (SFUs): For performing specialized operations such as trigonometric calculations.
2. **Caches**
    1. **L1 cache:** This cache reduces memory access latency, ensuring faster access to data required by the threads.
    2. **Shared memory:** Shared memory is a fast, low-latency memory that is shared among the threads in a block. It enables threads to collaborate efficiently by sharing data.
    3. **Constant cache:** Optimized for broadcasting data, the constant cache provides read-only memory for threads. It's used when many threads need to read the same data.
    4. **Texture cache.** This cache is optimized for spatial locality in texture memory, commonly used in graphics and image processing. Spatial locality means that when a thread accesses a certain memory location, it is likely that neighboring threads will access nearby memory locations.

    The texture cache takes advantage of this by fetching a block of memory around the requested data, thus reducing the number of separate memory requests. By fetching multiple nearby memory locations in a single request, the texture cache reduces the overhead of separate memory accesses, improving overall memory bandwidth utilization.

Streaming Multiprocessors

4. **Schedulers for warps**: Each SM has warp schedulers responsible for dispatching instructions to different warps based on scheduling policies. These policies ensure that warps are issued instructions in an efficient and fair manner to maximize GPU utilization.

5. **A substantial number of registers**. Since each SM may have hundreds or thousands of threads running concurrently, registers must store a large amount of per-thread data. This large pool of registers allows each thread to quickly access data without needing to use slower global memory.

Device Global Memory and Data Transfer

- In CUDA, host and devices have separate memory spaces.
- To execute a kernel on a device, the programmer needs to allocate global memory (device memory) on the device and transfer pertinent data from the host memory to the allocated device memory. **(Part 1)**
- After device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. **(Part 3)**

```
#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
}
```

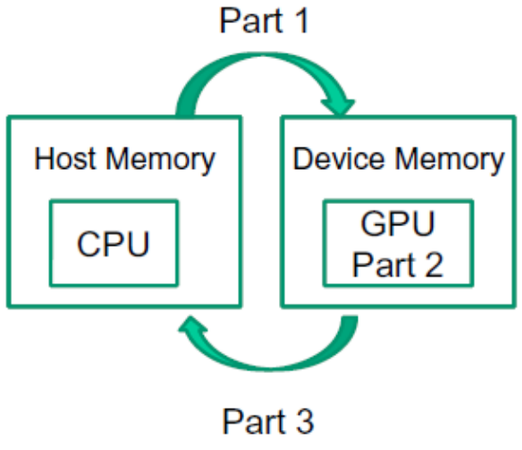


FIGURE 3.5

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

Device Global Memory and Data Transfer

- The CUDA runtime system provides API functions for managing data in the device memory.
- For example, Parts 1 and 3 of the vecAdd() function in Figure 3.5 need to use these API functions to allocate:
  - device memory for A, B, and C;
  - transfer A and B from host memory to device memory;
  - transfer C from device memory to host memory; and
  - free the device memory for A, B, and C.

Device Global Memory and Data Transfer

- 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.
- This parameter allows the **cudaMalloc()** function to write the address of the allocated memory into the pointer variable.
- The host code passes this pointer value to the kernels that need to access the allocated memory.
- The second parameter to the **cudaMalloc()** function gives the size of the data to be allocated, in terms of bytes.
- After the computation, **cudaFree()** is called with pointer **d_A** as input to free the storage space.

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

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 **cudaMemcpy()**.
- The **cudaMemcpy()** function takes four parameters.
- A pointer to the **destination location** for the data object to be copied.
- The second parameter points to the **source location**.
- The third parameter specifies the **number of bytes to be copied**.
- The fourth parameter indicates the **types of memory involved** in the copy:
  - from host memory to host memory
  - from host memory to device memory
  - from device memory to host memory
  - from device memory to device memory

cudaMemcpy()
- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

Device Global Memory and Data Transfer

```
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);
}
```

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.
- In CUDA, a __syncthreads() statement, if present, must be executed by all threads in a block
- When a __syncthread() statement is placed in an if statement, it must either be executed by all threads in the block or none of them.

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

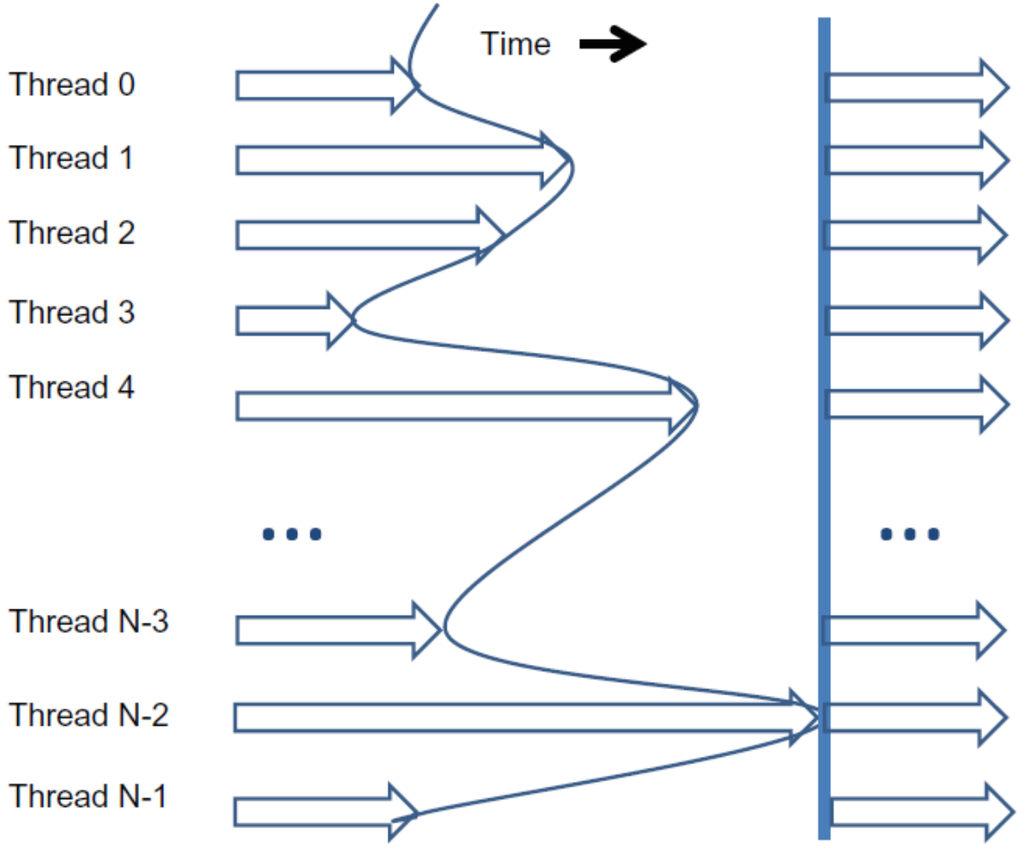


FIGURE 4.11

An example execution timing of barrier synchronization.

Synchronization and Transparent Scalability

- 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.
- In a typical if-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.
- 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
```

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.

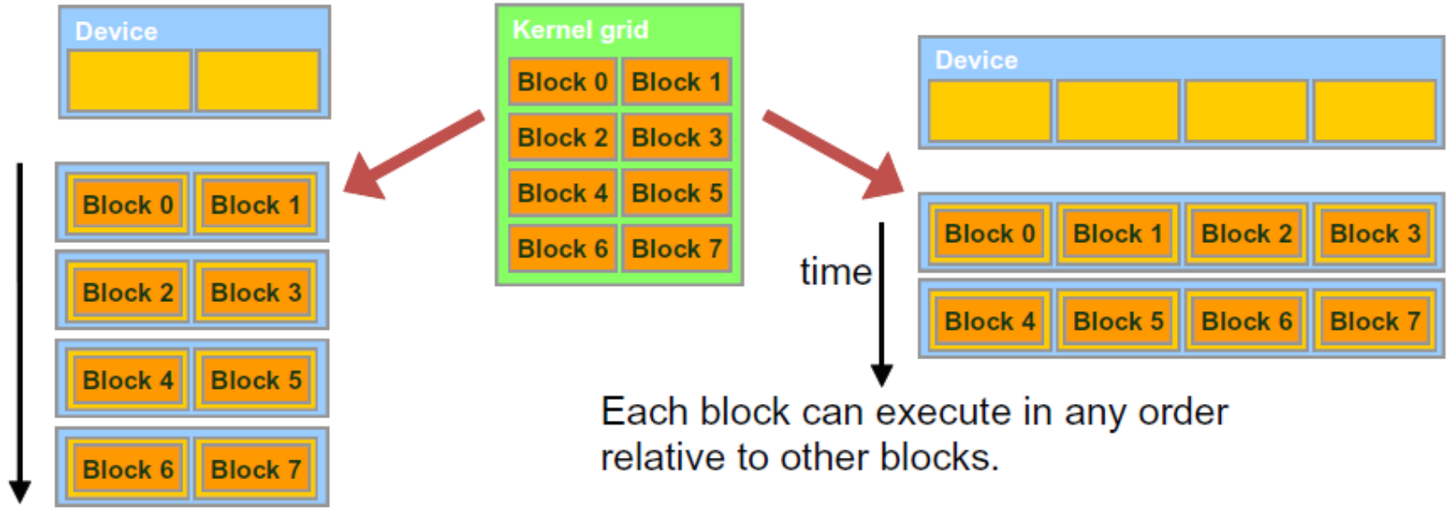


FIGURE 4.12

Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

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.
- Ex: A mobile processor may execute an application slowly but at extremely low power consumption, and a desktop processor may execute the same application at a higher speed while consuming more power.
- 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).
- Each device has a limit on the number of blocks that can be assigned to each SM.
- 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.

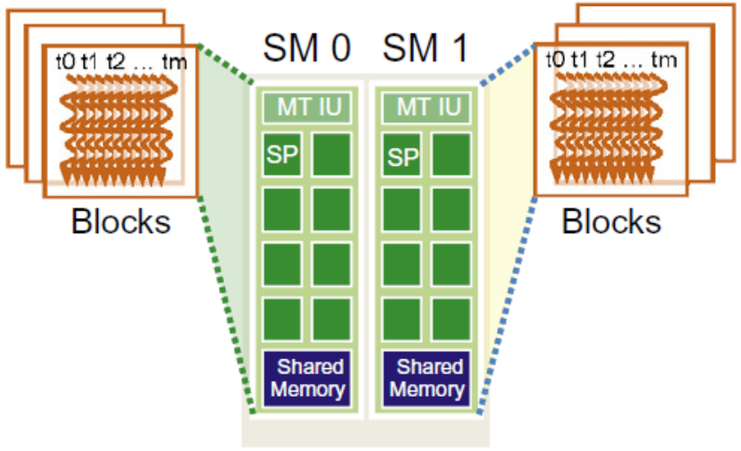


FIGURE 4.13

Thread block assignment to SMs.

Passing Parameter

- We can pass parameters to a kernel as we would with any C function
- We need to allocate memory to do anything useful on a device, such as return values to the host
- A kernel call looks and acts exactly like any function call in standard C
- The runtime system takes care of any complexity introduced by the fact that these parameters need to get from the host to the device

```
#include <iostream>
#include "book.h"

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

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c,
                              dev_c,
                              sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );

    return 0;
}
```

Passing Parameter

- The allocation of memory using **cudaMalloc()**
- This call behaves very similarly to the standard C call **malloc(),** but it tells the CUDA runtime to allocate the memory on the device
- The first argument is a pointer to the pointer you want to hold the address of the newly allocated memory
- The second parameter is the size of the allocation you want to make
- The HANDLE_ERROR() that surrounds these calls is a utility macro
  - It simply detects that the call has returned an error, prints the associated error message, and exits the application with an EXIT_FAILURE code

Passing Parameter

- it is the responsibility of the programmer not to dereference the pointer returned by **cudaMalloc()** from code that executes on the host
- Host code may pass this pointer around, perform arithmetic on it, or even cast it to a different type. But you cannot use it to read or write from memory

Passing Parameter

- Restrictions on the usage of device pointer as follows:
  - You can pass pointers allocated with cudaMalloc() to functions that execute on the device
  - You can use pointers allocated with cudaMalloc()to read or write memory from code that executes on the device
  - You can pass pointers allocated with cudaMalloc()to functions that execute on the host
  - You cannot use pointers allocated with cudaMalloc()to read or write memory from code that executes on the host.

Passing Parameter

- We can't use standard C's free() function to release memory we've allocated with **cudaMalloc()**
- To free memory we've allocated with **cudaMalloc()**, we need to use a call to **cudaFree()**
- Two of the most common methods for accessing device memory
  - by using device pointers from within device code
  - By using calls to **cudaMemcpy()**
- Host pointers can access memory from host code, and device pointers can access memory from device code

Passing Parameter

- we can also access memory on a device through calls to **cudaMemcpy()** from host code
- These calls behave exactly like standard C **memcpy()** with an additional parameter to specify which of the source and destination pointers point to device memory.
- The last parameter to **cudaMemcpy()** is **cudaMemcpyDeviceToHost**, instructing the runtime that the source pointer is a device pointer and the destination pointer is a host pointer
- **cudaMemcpyHostToDevice** would indicate the opposite situation, where the source data is on the host and the destination is an address on the device
- Finally, we can even specify that both pointers are on the device by passing **cudaMemcpyDeviceToDevice**

Querying Devices