BOOK 02: CHAPTER 3 INTRODUCTION TO DATA PARALLELISM AND CUDA C

OBJECTIVE

- The main objective is to teach the key concepts involved in writing massively parallel programs in a heterogeneous computing system.

INTRODUCTION

- CUDA C is an extension to the popular C programming language1 with new keywords and application programming interfaces for programmers to take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPU's.
- The computing system consists of a host that is a traditional CPU, such as an Intel architecture microprocessor in personal computers today, and one or more devices that are processors with a massive number of arithmetic units.
- A CUDA device is typically a GPU Graphics Processing Units.

DATA PARALLELISM

- Task parallelism is the simultaneous execution on multiple cores of many different functions across the same or different datasets.
- Data parallelism (aka SIMD) is the simultaneous execution on multiple cores of the same function across the elements of a dataset.
- Here are some reasons why data parallelism is used in CUDA C:
    - GPUs (Graphics Processing Units) are designed to handle a large number of parallel tasks simultaneously. They consist of thousands of small processing cores that can execute the same operation on different pieces of data concurrently.
    - Data parallelism leverages the massive parallelism of GPUs by dividing a large dataset into smaller chunks and processing these chunks simultaneously on different GPU cores.
    - SIMD (Single Instruction, Multiple Data) Architecture:
    - GPUs are equipped with a SIMD architecture, where a single instruction is applied to multiple data elements at the same time. This aligns well with data parallelism, where the same operation is performed on different elements of a dataset concurrently.
    - CUDA C allows programmers to express parallelism explicitly through the use of thread hierarchies, making it easier to write code that takes advantage of SIMD execution.

Data Parallelism with A Vector Addition Example

In this example, each element of the sum vector C is generated by adding an element of input vector A to an element of input vector B. For example, C[0] is generated by adding A[0] to B[0], and C[3] is generated by adding A[3] to B[3]. All additions can be performed in parallel. Therefore, vector addition of two large vectors exhibits a rich amount of data parallelism.

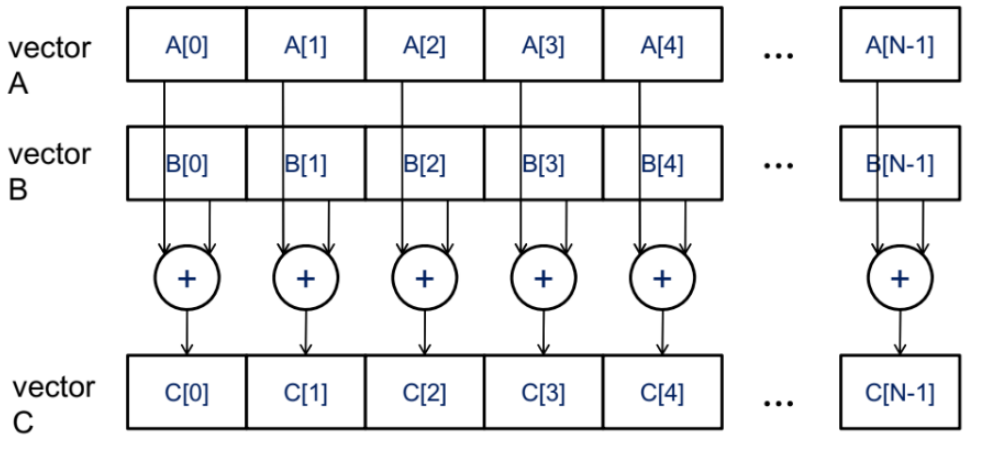

FIGURE 3.1

Data parallelism in vector addition.

CUDA PROGRAM STRUCTURE

- The structure of a CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer.
- Each CUDA source file can have a mixture of both host and device code.
- By default, any traditional C program is a CUDA program that contains only host code. One can add device functions and data declarations into any C source file.
- The function or data declarations for the device are clearly marked with special CUDA keywords.
- The CUDA keywords are used to separate the host code and device code.
- These are typically functions that exhibit a rich amount of data parallelism.
- The code needs to be compiled by a compiler that recognizes and understands these additional declarations such as CUDA C compiler by NVIDIA called NVCC (NVIDIA C Compiler).

Host Code

- The host code is straight ANSI C code, which is further compiled with the host's standard C/C++ compilers and is run as a traditional CPU process.

Device Code

- The device code is marked with CUDA keywords for labeling data-parallel functions, called kernels, and their associated data structures.
- The device code is further compiled by a runtime component of NVCC and executed on a GPU device.

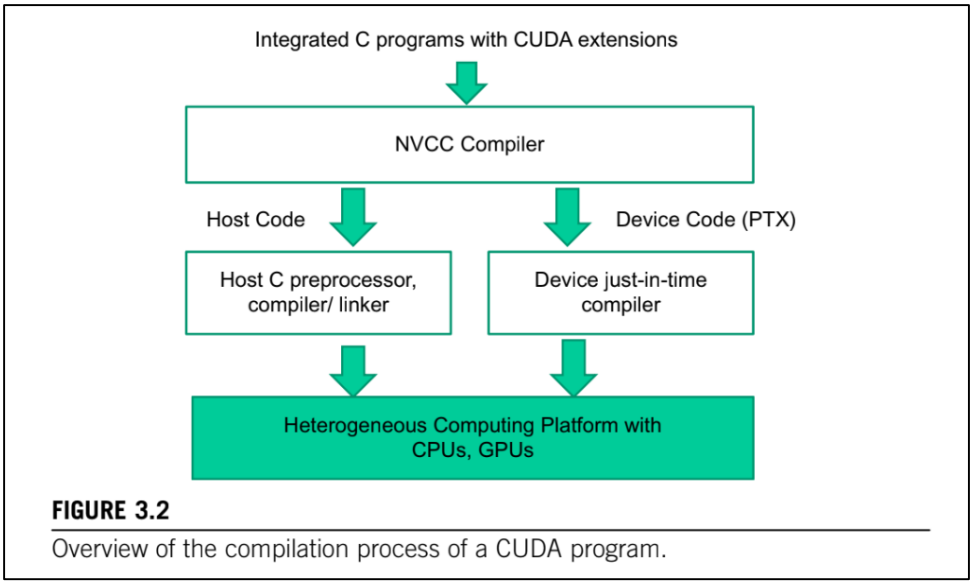

FIGURE 3.2

Overview of the compilation process of a CUDA program.

- The execution of a CUDA program is illustrated in Figure 3.3.
- The execution starts with host (CPU) execution. When a kernel function is called, or launched, it is executed by a large number of threads on a device.
- All the threads that are generated by a kernel launch are collectively called a grid.
- Figure 3.3 shows the execution of two grids of threads.
- When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is launched.

Note: Figure 3.3 shows a simplified model where the CPU execution and the GPU execution do not overlap. Many heterogeneous computing applications actually manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

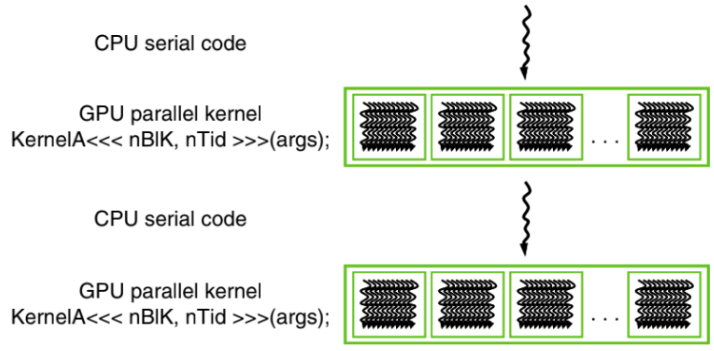

FIGURE 3.3
Execution of a CUDA program.

Execution Model

- A thread is a simplified view of how a processor executes a program in modern computers. A thread consists of the code of the program, the particular point in the code that is being executed, and the values of its variables and data structures. The execution of a thread is sequential.
- If we want to start parallel execution in an application, we need to create and manage multiple threads
- In CUDA, the execution of each thread is sequential as well. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying runtime mechanisms to create many threads that process different parts of the data in parallel.

Vector Addition Example:

- In the vector addition example, each thread can be used to compute one element of the output vector C. In this case, the number of threads that will be generated by the kernel is equal to the vector length. For long vectors, a large number of threads will be generated. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support. This is in contrast with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

A VECTOR ADDITION KERNEL

- We now use vector addition to illustrate the CUDA programming model. we show the kernel code for vector addition.

Conventional CPU-Only Vector Addition

- Figure 3.4 shows a simple traditional C program that consists of a main function and a vector addition function.
- We will prefix the names of variables that are mainly processed by the host with h_ and those of variables that are mainly processed by a device d_.
- Assume that the vectors to be added are stored in arrays h_A and h_B that are allocated and initialized in the main program. The output vector is in array h_C, which is also initialized in the main program.
- The pointers to these arrays are passed to the vecAdd() function, along with the variable N that contains the length of the vectors.

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

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

FIGURE 3.4

A simple traditional vector addition C code example.

CUDA C Vector Addition

- A straightforward way to execute vector addition in parallel is to modify the vecAdd() function and move its calculations to a CUDA device.
- The structure of such a modified vecAdd() function is shown in Figure 3.5.
- At the beginning of the file, we need to add a C preprocessor directive to include the CUDA.h header file. This file defines the CUDA API functions and built-in variables.

- Part 1 of the function allocates space in the device (GPU) memory to hold copies of the A, B, and C vectors, and copies the vectors from the host memory to the device memory.
- Part 2 launches parallel execution of the actual vector addition kernel on the device.
- Part 3 copies the sum vector C from the device memory back to the host memory.
- vecAdd() function is essentially an outsourcing agent that ships input data to a device, activates the calculation on the device, and collects the results from the device.

```
#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
}
```

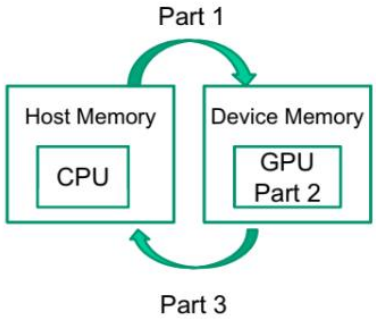

FIGURE 3.5

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

Part 1 allocates device memory for d_A, d_B, and d_C and transfers A to d_A and B to d_B. This is done by calling the cudaMalloc() and cudaMemcpy() functions.

Part 2 invokes the kernel.

Part 3 copies the sum data from device memory to host memory so that the value will be available to main(). This is accomplished with a call to the cudaMemcpy() function. It then frees the memory for d_A, d_B, and d_C from the device memory, which is done by calls to the cudaFree() function.

DEVICE GLOBAL MEMORY AND DATA TRANSFER

In CUDA, host and devices have separate memory spaces.

Devices are often hardware cards that come with their own DRAM called global memory/device memory.

To execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory (Part 1 of Figure 3.5).

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 of Figure 3.5).

The CUDA runtime system provides Application Programming Interface (API) functions to perform these activities on behalf of the programmer.

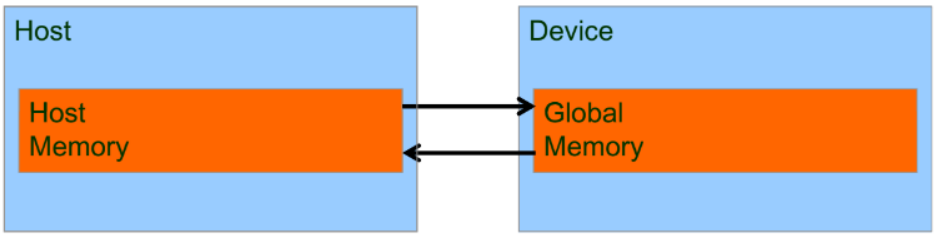

FIGURE 3.6

Host memory and device global memory.

Figure 3.6 shows a CUDA host memory and device memory model for programmers to reason about the allocation of device memory and movement of memory between host and device.

The device global memory can be accessed by the host to transfer data to and from the device between these memories and the host.

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.

Memory Allocation and Free Functions

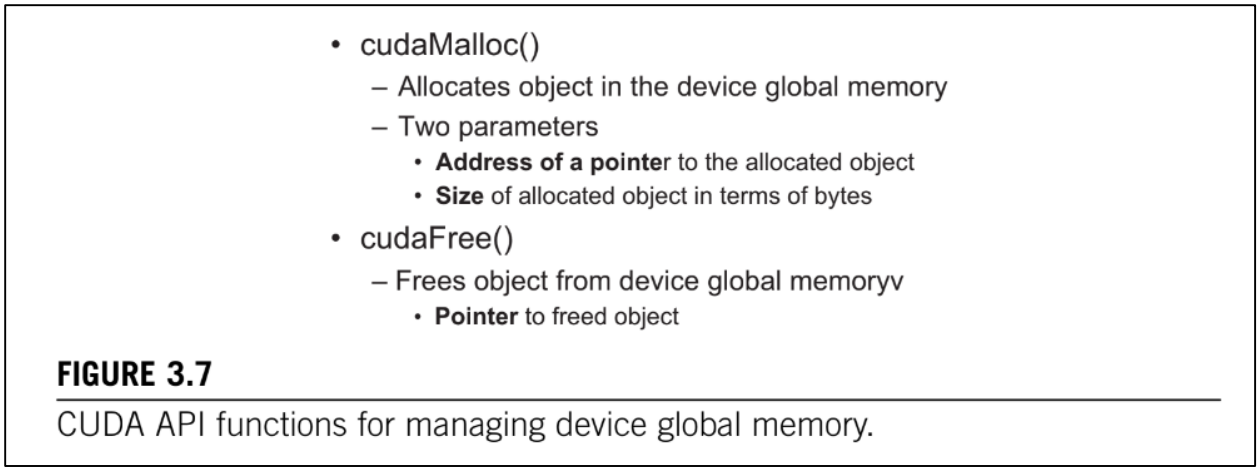

FIGURE 3.7

CUDA API functions for managing device global memory.

Figure 3.7 shows two API functions for allocating and freeing device global memory.

Function cudaMalloc() can be called from the host code to allocate a piece of device global memory for an object.

The first parameter to the cudaMalloc() function is the address of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to (void) because the function expects a generic pointer. The host code passes this pointer value to the kernels that need to access the allocated memory object.

The second parameter to the cudaMalloc() function gives the size of the data to be allocated, in terms of bytes.

Example To Illustrate The Use Of Cudamalloc()

The program passes the address of d_A (i.e., &d_A) as the first parameter after casting it to a void pointer. That is, d_A will point to the device memory region allocated for the A vector. The size of the allocated region will be n times the size of a single-precision floating number.

After the computation, cudaFree() is called with pointer d_A as input to free the storage space for the A vector from the device global memory.

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

The addresses in d_A, d_B, and d_C are addresses in the device memory. These addresses should not be dereferenced in the host code. They should be mostly used in calling API functions and kernel functions. Dereferencing a device memory point in the host code can cause exceptions or other types of runtime errors during runtime.

CUDA API Function for Data Transfer Between Host and Device

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.

- The first parameter is 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, and from device memory to device memory.

For example, the cudaMemcpy() function can be used to copy data from one location of the device memory to another location of the device memory.

Note: cudaMemcpy() cannot be used to copy between different GPUs in multi-GPU systems.

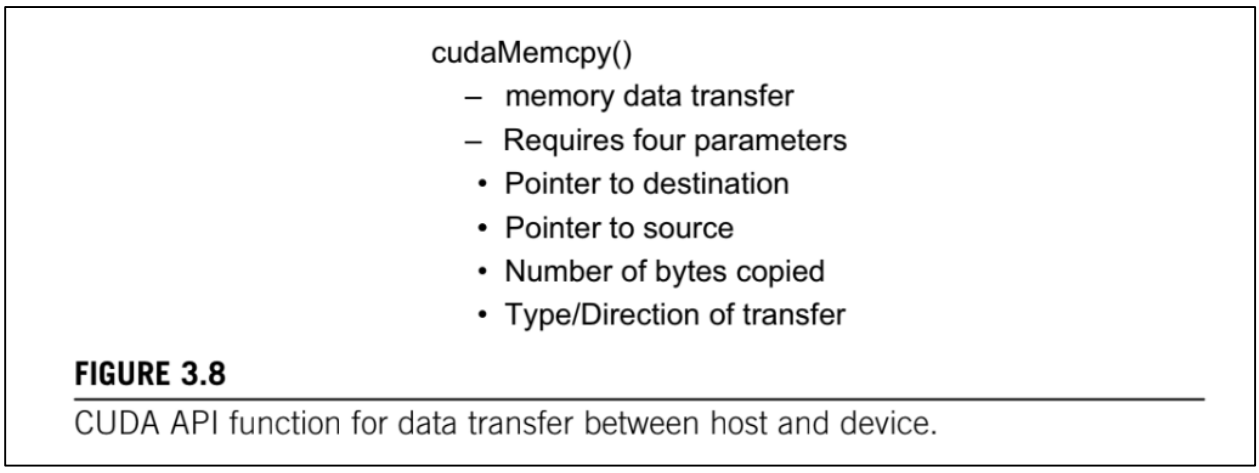

FIGURE 3.8

CUDA API function for data transfer between host and device.

A More Complete Version Of Vecadd()

The vecAdd() function calls the cudaMemcpy() function to copy A and B vectors from host to device before adding them and to copy the C vector from the device to host after the addition is done. Assume that the value of A, B, d_A, d_B, and size have already been set as we discussed before; the three cudaMemcpy() calls are shown below. The two symbolic constants, cudaMemcopyHostToDevice and cudaMemcopyDeviceToHost, are recognized, predefined constants of the CUDA programming environment.

Note that the same function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type.

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

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

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

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

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

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

FIGURE 3.9

A more complete version of `vecAdd()`.

The vecAdd() function allocates device memory, requests data transfers, and launches the kernel that performs the actual vector addition. We often refer to this type of host code as a stub function for launching a kernel. After the kernel finishes execution, vecAdd() also copies result data from device to the host.

ERROR HANDLING IN CUDA

In general, it is very important for a program to check and handle errors. CUDA API functions return flags that indicate whether an error has occurred when they served the request. Most errors are due to inappropriate argument values used in the call.

For brevity, we will not show error checking code in our examples. For example, line 1 in Figure 3.9 shows a call to `cudaMalloc()`:

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

In practice, we should surround the call with code that tests for error conditions and prints out error messages so that the user can be aware of the fact that an error has occurred. A simple version of such checking code is as follows:

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
printf("%s in %s at line %d\n", cudaGetErrorString( err),
  __FILE__, __LINE__);
exit(EXIT_FAILURE);
}
```

This way, if the system is out of device memory, the user will be informed about the situation.

One would usually define a C macro to make the checking code more concise in the source.

KERNEL FUNCTIONS AND THREADING

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

Note that SPMD is not the same as SIMD (single instruction, multiple data). In an SPMD system, the parallel processing units execute the same program on multiple parts of the data. However, these processing units do not need to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.

When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy.

Each grid is organized into an array of thread blocks, which will be referred to as blocks. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.

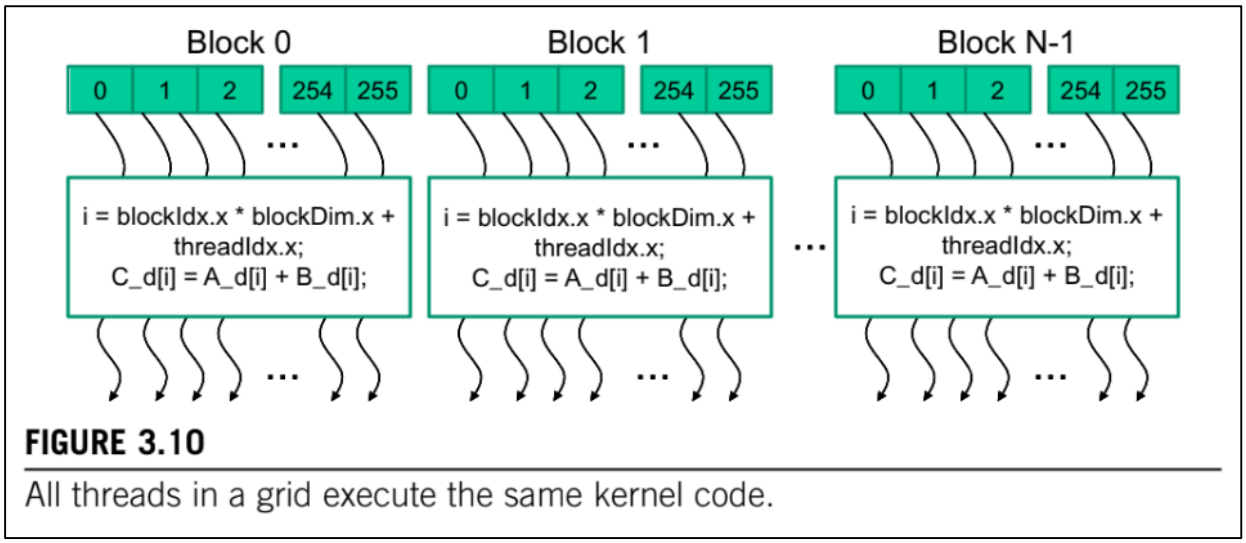

FIGURE 3.10

All threads in a grid execute the same kernel code.

Figure 3.10 shows an example where each block consists of 256 threads. The number of threads in each thread block is specified by the host code when a kernel is launched. The same kernel can be launched with different numbers of threads at different parts of the host code.

For a given grid of threads, the number of threads in a block is available in the blockDim variable. In Figure 3.10, the value of the blockDim.x variable is 256.

Each thread in a block has a unique threadIdx value. For example, the first thread in block 0 has value 0 in its threadIdx variable, the second thread has value 1, the third thread has value 2, etc.

This allows each thread to combine its threadIdx and blockIdx values to create a unique global index for itself with the entire grid.

In Figure 3.10, a data index i is calculated as i 5 blockIdx.x blockDim.x 1 threadIdx.x. Since blockDim is 256 in our example, the i values of threads in block 0 ranges from 0 to 255.

The i values of threads in block 1 range from 256 to 511. The i values of threads in block 2 range from 512 to 767. That is, the i values of the threads in these three blocks form a continuous coverage of the values from 0 to 767.

Since each thread uses i to access d_A, d_B, and d_C, these threads cover the first 768 iterations of the original loop. By launching the kernel with a larger number of blocks, one can process

larger vectors. By launching a kernel with n or more threads, one can process vectors of length n.

The Key Words Threadidx.X. Blockidx.X, And Blockdim.X

Note that all threads execute the same kernel code. There needs to be a way for them to distinguish among themselves and direct each thread toward a particular part of the data. These keywords identify predefined variables that correspond to hardware registers that provide the identifying coordinates to threads.

Different threads will see different values in their threadIdx.x, blockIdx.x, and blockDim.x variables.

The Automatic Variable i

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

```
// 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.

Figure 3.11 shows a kernel function for a vector addition. First, there is a CUDA specific keyword __global__ in front of the declaration of vecAddKernel(). This keyword indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.

	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.

The __global__ keyword indicates that the function being declared is a CUDA kernel function. Note that there are two underscore characters on each side of the word "global." A __global__ function is to be executed on the device and can only be called from the host code.

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 or another device 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. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration.

Note that one can use both __host__ and __device__ in a function declaration. This combination tells the compilation system to generate two versions of object files for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function.

When the host code launches a kernel, it sets the grid and thread block dimensions via execution configuration parameters. This is illustrated in Figure 3.13.

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

FIGURE 3.13

A vector addition kernel function and its launch statement.

The configuration parameters are given between the <<<< and >>>> before the traditional C function arguments.

The first configuration parameter gives the number of thread blocks in the grid.

The second specifies the number of threads in each thread block.

In this example, there are 256 threads in each block. To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1,000 threads, we would launch ceil(1,000/256.0) 5 4 thread blocks. As a result, the statement will launch 4 3 256 5 1,024 threads.

Figure 3.14 shows the final host code of vecAdd().

```
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);

    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

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

FIGURE 3.14

A complete version of `vecAdd()`.