CHAPTER

Introduction to Data Parallelism and CUDA C

3

CHAPTER OUTLINE

Our main objective is to teach the key concepts involved in writing massively parallel programs in a heterogeneous computing system. This requires many code examples expressed in a reasonably simple language that supports massive parallelism and heterogeneous computing. We have chosen CUDA C for our code examples and exercises. CUDA C is an extension to the popular C programming language[1] 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. For the rest of this book, we will refer to CUDA C simply as CUDA. To a CUDA programmer, 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. Many modern software applications have sections that exhibit a rich amount of data parallelism, a phenomenon that allows arithmetic operations

[1]CUDA C also supports a growing subset of C++ features. Interested readers should refer to the *CUDA Programming Guide* for more information about the supported C++ features.

to be safely performed on different parts of the data structures in parallel. CUDA devices accelerate the execution of these applications by applying their massive number of arithmetic units to these data-parallel program sections. Since data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

3.1 DATA PARALLELISM

Modern software applications often process a large amount of data and incur long execution time on sequential computers. Many of them operate on data that represents or models real-world, physical phenomena. Images and video frames are snapshots of a physical world where different parts of a picture capture simultaneous, independent physical events. Rigid-body physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps. Airline scheduling deals with thousands of flights, crews, and airport gates that operate in parallel. Such independent evaluation is the basis of data parallelism in these applications.

TASK PARALLELISM VERSUS DATA PARALLELISM

Data parallelism is not the only type of parallelism widely used in parallel programming. Task parallelism has also been used extensively in parallel programming. Task parallelism is typically exposed through task decomposition of applications. For example, a simple application may need to do a vector addition and a matrix–vector multiplication. Each of these would be a task. Task parallelism exists if the two tasks can be done independently.

In large applications, there are usually a larger number of independent tasks and therefore a larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks includes vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce CUDA streams.

Let us illustrate the concept of data parallelism with a vector addition example in Figure 3.1. 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

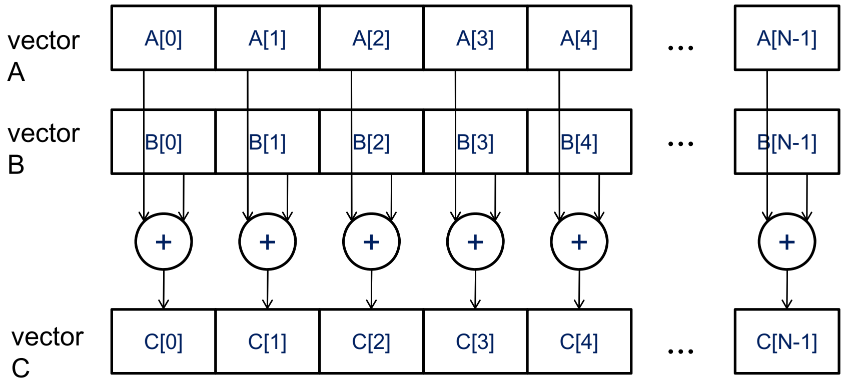

FIGURE 3.1

Data parallelism in vector addition.

parallel. Therefore, vector addition of two large vectors exhibits a rich amount of data parallelism. Data parallelism in real applications can be more complex and will be discussed in detail later.

3.2 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. These are typically functions that exhibit a rich amount of data parallelism.

Once device functions and data declarations are added to a source file, it is no longer acceptable to a traditional C compiler. The code needs to be compiled by a compiler that recognizes and understands these additional declarations. We will be using a CUDA C compiler by NVIDIA called NVCC (NVIDIA C Compiler). As shown at the top of Figure 3.2, the NVCC processes a CUDA program, using the CUDA keywords to separate the host code and device 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. 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

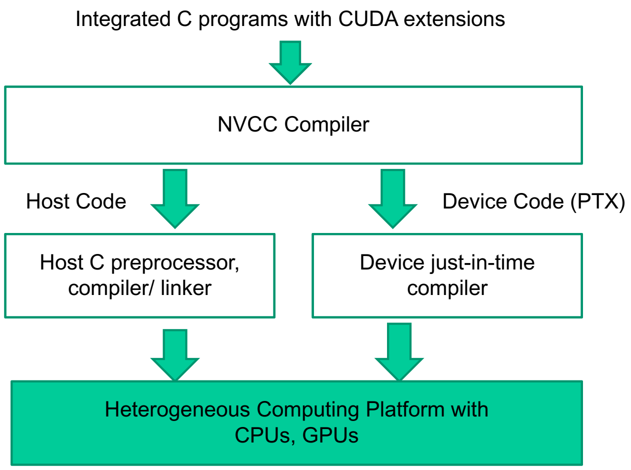
Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

FIGURE 3.2

Overview of the compilation process of a CUDA program.

executed on a GPU device. In situations where there is no device available or a kernel can be appropriately executed on a CPU, one can also choose to execute the kernel on a CPU using tools like MCUDA [Stratton 2008].

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. We will discuss how these grids are organized soon. 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 that 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.

THREADS

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 as far as a user is concerned. One can use a source-level debugger to monitor the progress of a

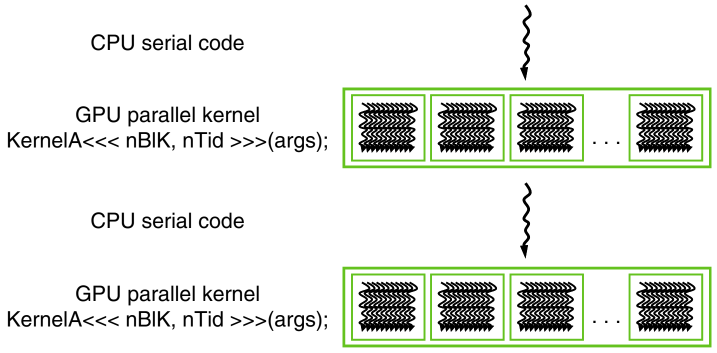

FIGURE 3.3

Execution of a CUDA program.

thread by executing one statement at a time, looking at the statement that will be executed next, and checking the values of the variables and data structures.

Threads have been used in traditional CPU programming for many years. If a programmer wants to start parallel execution in an application, he or she needs to create and manage multiple threads using thread libraries or special languages.

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.

Launching a kernel typically generates a large number of threads to exploit data parallelism. 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.

3.3 A VECTOR ADDITION KERNEL

We now use vector addition to illustrate the CUDA programming model. Before we show the kernel code for vector addition, it is helpful to first review how a conventional CPU-only vector addition function works. Figure 3.4 shows a simple traditional C program that consists of a main function and a vector addition function. In each piece of host code, we will prefix the names of variables that are mainly processed by the host

```
// 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.

with h_ and those of variables that are mainly processed by a device d_ to remind ourselves the intended usage of these variables.

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. For brevity, we do not show the details of how h_A, V, and h_C are allocated or initialized. A complete source code listing that contains more details is available in Appendix A. The pointers to these arrays are passed to the vecAdd() function, along with the variable N that contains the length of the vectors.

POINTERS IN THE C LANGUAGE

The function arguments A, B, and C in Figure 3.4 are pointers. In the C language, a pointer can be used to access variables and data structures. While a floating point variable V can be declared wit:
float V;
a pointer variable P can be declared with:
float *P;
By assigning the address of V to P with the statement P = & V, we make P “point to” V. *P becomes a synonym for V. For example U = *P assigns the value of V to U. For another example, *P = 3 changes the value of V to 3. An array in a C program can be accessed through a pointer that points to its 0th element. For example, the statement P = & (h_A[0]) makes P point to the 0th element of array h_A. P[i] becomes a synonym for h_A[i]. In fact, the array name h_A is in itself a pointer to its 0th element. In Figure 3.4, passing an array name h_A as the first argument to function call to vecAdd makes the function’s first parameter A point to the 0th element of h_A . We say that h_A is passed by reference to vecAdd. As a result, A [i] in the function body can be used to access h_A[i]. See Patt& Patel [Patt] for an easy-to-follow explanation of the detailed usage of pointers in C.

```
#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
}
```

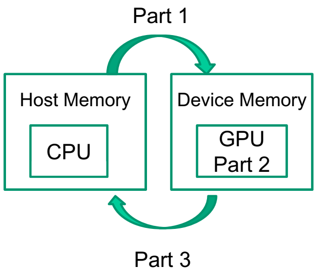

FIGURE 3.5

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

The `vecAdd()` function in Figure 3.4 uses a `for` loop to iterate through the vector elements. In the *i*th iteration, output element `C[i]` receives the sum of `A[i]` and `B[i]`. The vector length parameter `n` is used to control the loop so that the number of iterations matches the length of the vectors. The parameters `A`, `B`, and `C` are passed by reference so the function reads the elements of `h_A`, `h_B` and writes the elements of `h_C` through the parameter pointers `A`, `B`, and `C`. When the `vecAdd()` function returns, the subsequent statements in the main function can access the new contents of `h_C`.

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 that we will be introducing soon. 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.

Note that the revised `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. The agent does so in such a way that the main program does not need to even be aware that the vector addition is now actually done on a device. The details of the revised

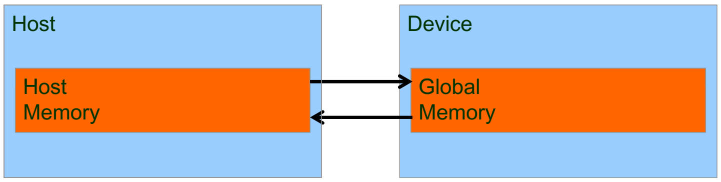

FIGURE 3.6

Host memory and device global memory.

function, as well as the way to compose the kernel function, will be shown as we introduce the basic features of the CUDA programming model.

3.4 DEVICE GLOBAL MEMORY AND DATA TRANSFER

In CUDA, host and devices have separate memory spaces. This reflects the current reality that devices are often hardware cards that come with their own DRAM. For example, the NVIDIA GTX480 comes with up to 4 GB[2] (billion bytes, or gigabytes) of DRAM, called global memory. We will also refer to global memory as 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. This corresponds to Part 1 of Figure 3.5. Similarly, 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. This corresponds to 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. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory. The same holds for the opposite direction.

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, as illustrated by the bidirectional arrows between these memories and the

[2]There is a trend to integrate CPUs and GPUs into the same chip package, commonly referred to as *fusion*. Fusion architectures often have a unified memory space for host and devices. There are new programming frameworks, such as GMAC, that take advantage of the unified memory space and eliminate data copying costs.

host in Figure 3.6. There are more device memory types than shown in Figure 3.6. Constant memory can be accessed in a read-only manner by device functions, which will be described in Chapter 8. We will also discuss the use of registers and shared memory in Chapter 5. See the *CUDA Programming Guide* for the functionality of texture memory. For now, we will focus on the use of global memory.

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. We will explain the memory allocation and free functions first. 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. Readers should notice the striking similarity between `cudaMalloc()` and the standard C runtime library `malloc()`. This is intentional; CUDA is C with minimal extensions. 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. By keeping the interface as close to the original C runtime libraries as possible, CUDA minimizes the time that a C programmer spends to relearn the use of these extensions.

The first parameter to the `cudaMalloc()` function is the address of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to `(void **)` because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.[3] This parameter allows the `cudaMalloc()` function to write the address of the allocated memory into the pointer variable.[4] The host code passes this pointer value to the kernels that need to access the allocated memory

[3]The fact that `cudaMalloc()` returns a generic object makes the use of dynamically allocated multidimensional arrays more complex. We will address this issue in Section 4.2.

[4]Note that `cudaMalloc()` has a different format from the C `malloc()` function. The C `malloc()` function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The `cudaMalloc()` function writes to the pointer variable of which the address is given as the first parameter. As a result, the `cudaMalloc()` function takes two parameters. The two-parameter format of `cudaMalloc()` allows it to use the return value to report any errors in the same way as other CUDA API functions.

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

FIGURE 3.7

CUDA API functions for managing device global memory.

object. The second parameter to the `cudaMalloc()` function gives the size of the data to be allocated, in terms of bytes. The usage of this second parameter is consistent with the size parameter to the C `malloc()` function.

We now use a simple code example to illustrate the use of `cudaMalloc()`. This is a continuation of the example in Figure 3.5. For clarity, we will start a pointer variable with `d_` to indicate that it points to an object in the device memory. The program passes the address of `d_A` (i.e., `&d_A`) as the first parameter after casting it to a void pointer. That is, `d_A` will point to the device memory region allocated for the A vector. The size of the allocated region will be *n* times the size of a single-precision floating number, which is 4 bytes in most computers today. After the computation, `cudaFree()` is called with pointer `d_A` as input to free the storage space for the A vector from the device global memory.

```
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.

Readers should complete Part 1 of the `vecAdd()` example in Figure 3.5 with similar declarations of `d_B` and `d_C` pointer variables as well as their corresponding `cudaMalloc()` calls. Furthermore, Part 3 in Figure 3.6 can be completed with the `cudaFree()` calls for `d_B` and `d_C`.

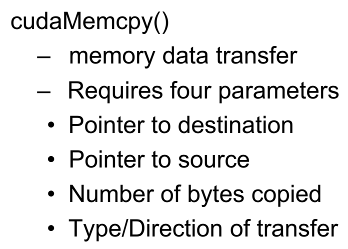
cudaMemcpy()
- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

FIGURE 3.8

CUDA API function for data transfer between host and device.

Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions. Figure 3.8 shows such an API function, `cudaMemcpy()`. 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.[5]

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);
```

[5]Please note `cudaMemcpy()` cannot be used to copy between different GPUs in multi-GPU systems.

```
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()`.

```
    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.

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.

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

To summarize, the main program in Figure 3.4 calls `vecAdd()`, which is also executed on the host. The `vecAdd()` function, outlined in Figure 3.5, 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. We show a more complete version of the `vecAdd()` function in Figure 3.9.

Compared to Figure 3.6, the `vecAdd()` function in Figure 3.9 is complete for Parts 1 and 3. 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. Readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Figure 3.9. Part 2 invokes the kernel and will be described in the following section. 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.

3.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of launching these kernel functions. 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) [Atallah1998] parallel programming style, a popular programming style for massively parallel computing systems.[6]

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

[6]Note that SPMD is not the same as SIMD (single instruction, multiple data) [Flynn1972]. 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.

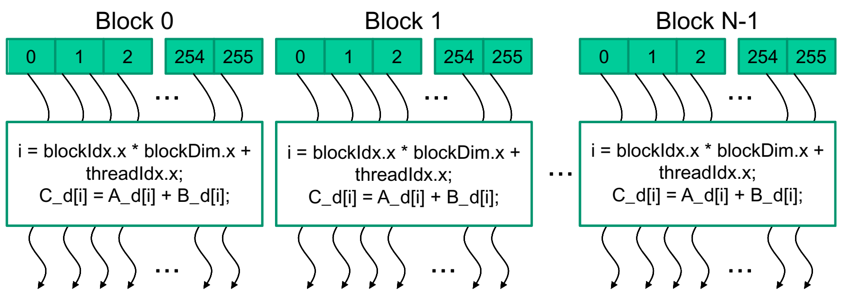
Block 0
0 1 2 … 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
Block 1
0 1 2 … 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
…
Block N-1
0 1 2 … 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

FIGURE 3.10

All threads in a grid execute the same kernel code.

grid is organized into an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.[7] 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. In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons. We will revisit this later.

Each thread in a block has a unique `threadIdx` value. For example, the first thread in block 0 has value 0 in its `threadIdx` variable, the second thread has value 1, the third thread has value 2, etc. This allows each thread to combine its `threadIdx` and `blockIdx` values to create a unique global index for itself with the entire grid. In Figure 3.10, a data index `i` is calculated as `i` = `blockIdx.x` * `blockDim.x` + `threadIdx.x`. Since `blockDim` is 256 in our example, the `i` values of threads in block 0 ranges from 0 to 255. The `i` values of threads in block 1 range from 256 to 511. The `i` values of threads in block 2 range from 512 to 767. 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*.

[7]Each thread block can have up to 1,024 threads in CUDA 3.0 and later. Some earlier CUDA versions allow only up to 512 threads in a block.

```
// 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.

	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.

Figure 3.11 shows a kernel function for a vector addition. The syntax is ANSI C with some notable extensions. 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.

In general, CUDA extends C language with three qualifier keywords in function declarations. The meaning of these keywords is summarized in Figure 3.12. 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.[8]

[8]We will explain the rules for using indirect function calls and recursions in different generations of CUDA later. In general, one should avoid the use of recursion and indirect function calls in their device functions and kernel functions to allow maximal portability.

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. This makes sense since many CUDA applications are ported from CPU-only execution environments. The programmer would add kernel functions and device functions during the porting process. The original functions remain as host functions. Having all functions to default into host functions spares the programmer the tedious work to change all original function declarations.

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. This supports a common-use case when the same function source code can be recompiled to generate a device version. Many user library functions will likely fall into this category.

The second notable extension to ANSI C in Figure 3.10 is the keywords `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. For simplicity, we will refer to a thread as $\text{thread}_{\text{blockIdx.x, threadIdx.x}}$. Note that the `.x` implies that there might be `.y` and `.z`. We will come back to this point soon.

There is an automatic (local) variable `i` in Figure 3.11. In a CUDA kernel function, automatic 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. The value assigned by a thread to its `i` variable is not visible to other threads. We will discuss these automatic variables again in Chapter 5.

A quick comparison between Figure 3.4 and Figure 3.11 reveals an important insight for CUDA kernels and a CUDA kernel launch. The kernel function in Figure 3.11 does not have a loop that corresponds to the one in Figure 3.4. Readers should ask where the loop went. The answer is that the loop is now replaced with the grid of threads. The entire

```
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.

grid forms the equivalent of the loop. Each thread in the grid corresponds to one iteration of the original loop.

Note that there is an `if (i<n)` statement in `addVecKernel()` in Figure 3.11. This is because not all vector lengths can be expressed as multiples of the block size. For example, if the vector length is 100, the smallest efficient thread block dimension is 32. Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their `i` values against `n`, which is 100. With the `if (i<n)` statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration* parameters. This is illustrated in Figure 3.13. The configuration parameters are given between the <<< and >>> before the traditional C function arguments. The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block. In this example, there are 256 threads in each block. To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to `n/256.0`. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1,000 threads, we would launch ceil(1,000/256.0) = 4 thread blocks. As a result, the statement will launch $4 \times 256 = 1{,}024$ threads. With the `if (i < n)` statement in the kernel as shown in Figure 3.11, the first 1,000 threads will perform addition on the 1,000 vector elements. The remaining 24 will not.

Figure 3.14 shows the final host code of `vecAdd()`. This source code completes the skeleton in Figure 3.5. Figures 3.10 and 3.14 jointly illustrate a simple CUDA program that consists of both the host code and a device

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

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

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

     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()`.

kernel. The code is hardwired to use thread blocks of 256 threads each. The number of thread blocks used, however, depends on the length of the vectors (`n`). If `n` is 750, three thread blocks will be used; if `n` is 4,000, 16 thread blocks will be used; if `n` is 2,000,000, 7,813 blocks will be used. Note that all the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order. A small GPU with a small amount of execution resources may execute one or two of these thread blocks in parallel. A larger GPU may execute 64 or 128 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware. That is, same code runs at lower performance on small GPUs and higher performance on larger GPUs. We will revisit this point again in Chapter 4.

It is important to point out that the vector addition example is used for its simplicity. In practice, the overhead of allocating device memory, input data transfer from host to device, output data transfer from device to host, and de-allocating device memory will likely make the resulting code slower than the original sequential code in Figure 3.4. This is because the amount of calculation done by the kernel is small relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand. Real applications typically have kernels where much more work is needed relative to the amount of data processed, which makes the additional overhead worthwhile. They also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized. We will present several examples of such applications.

3.6 SUMMARY

This chapter provided a quick overview of the CUDA programming model. CUDA extends the C language to support parallel computing. We discussed a subset of these extensions in this chapter. For your convenience, we summarize the extensions that we have discussed in this chapter as follows.

Function Declarations

CUDA extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Figure 3.12. Using one of `__global__`, `__device__`, or `__host__`, a CUDA programmer can instruct the compiler to generate a kernel function, a device function, or a host function. All function declarations without any of these keywords are defaulted to host functions. If both `__host__` and `__device__` are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA extension keyword, the function defaults into a host function.

Kernel Launch

CUDA extends C function call syntax with kernel execution configuration parameters surrounded by <<< and >>>. These execution configuration parameters are only used during a call to a kernel function, or a kernel launch. We discussed the execution configuration parameters that define the dimensions of the grid and the dimensions of each block. Readers should refer to the *CUDA Programming Guide* [NVIDIA2011] for more details of the kernel launch extensions as well as other types of execution configuration parameters.

Predefined Variables

CUDA kernels can access a set of predefined variables that allow each thread to distinguish among themselves and to determine the area of data each thread is to work on. We discussed the `threadIdx`, `blockDim`, and