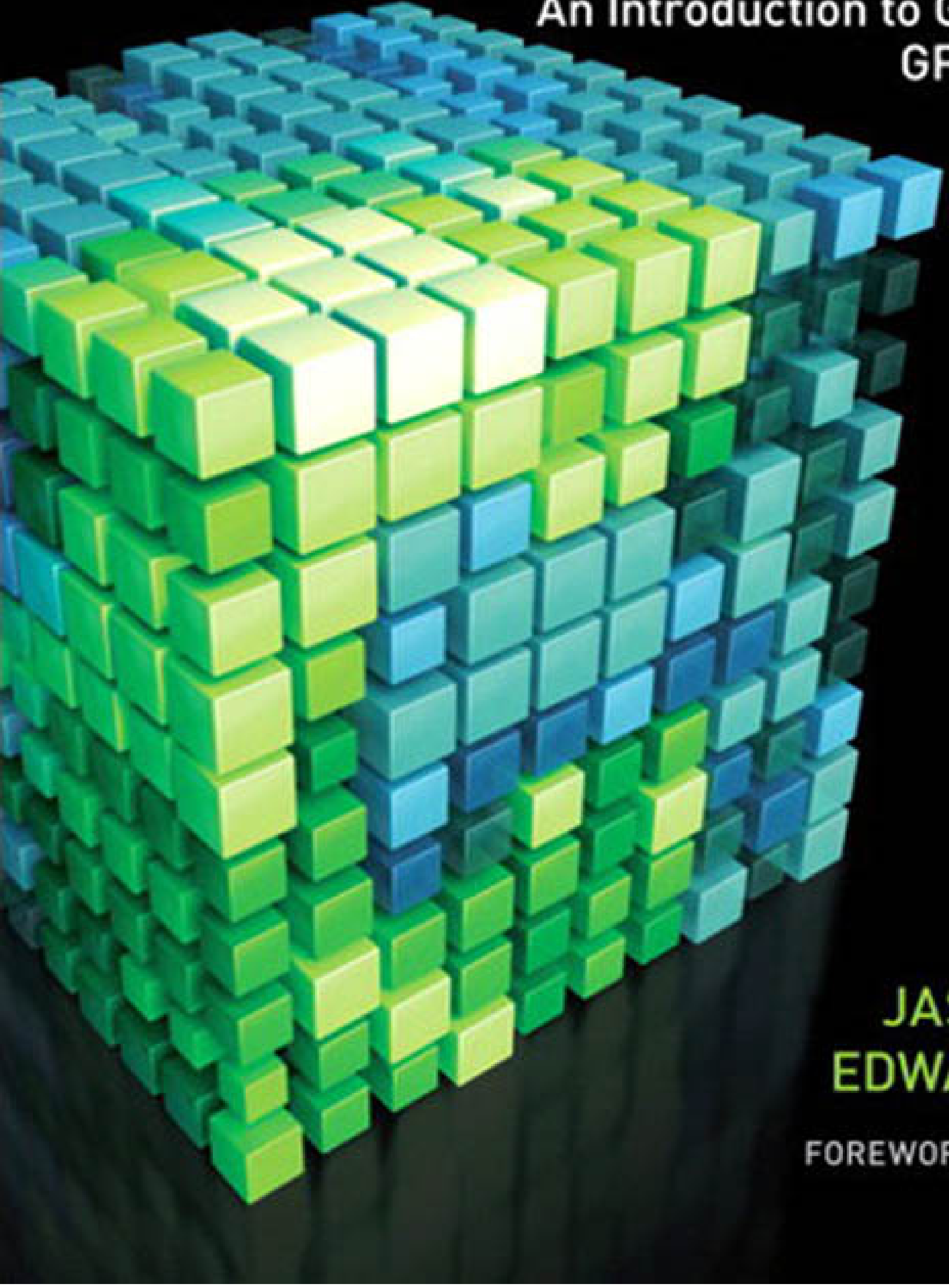
CUDA BY EXAMPLE

An Introduction to General-Purpose GPU Programming



JASON SANDERS
EDWARD KANDROT

FOREWORD BY JACK DONGARRA

Chapter 3

Introduction to CUDA C

If you read Chapter 1, we hope we have convinced you of both the immense computational power of graphics processors and that you are just the programmer to harness it. And if you continued through Chapter 2, you should have a functioning environment set up in order to compile and run the code you'll be writing in CUDA C. If you skipped the first chapters, perhaps you're just skimming for code samples, perhaps you randomly opened to this page while browsing at a bookstore, or maybe you're just dying to get started; that's OK, too (we won't tell). Either way, you're ready to get started with the first code examples, so let's go.

.1 Chapter Objectives

Through the course of this chapter, you will accomplish the following:

- You will write your first lines of code in CUDA C.
- You will learn the difference between code written for the *host* and code written for a *device*.
- You will learn how to run device code from the host.
- You will learn about the ways device memory can be used on CUDA-capable devices.
- You will learn how to query your system for information on its CUDA-capable devices.

.2 A First Program

Since we intend to learn CUDA C by example, let's take a look at our first example of CUDA C. In accordance with the laws governing written works of computer programming, we begin by examining a "Hello, World!" example.

3.2.1 HELLO, WORLD!

```
#include "../common/book.h"

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

At this point, no doubt you're wondering whether this book is a scam. Is this just C? Does CUDA C even exist? The answers to these questions are both in the affirmative; this book is not an elaborate ruse. This simple "Hello, World!" example is meant to illustrate that, at its most basic, there is no difference between CUDA C and the standard C to which you have grown accustomed.

The simplicity of this example stems from the fact that it runs entirely on the *host*. This will be one of the important distinctions made in this book; we refer to the CPU and the system's memory as the *host* and refer to the GPU and its memory as the *device*. This example resembles almost all the code you have ever written because it simply ignores any computing devices outside the host.

To remedy that sinking feeling that you've invested in nothing more than an expensive collection of trivialities, we will gradually build upon this simple example. Let's look at something that uses the GPU (a *device*) to execute code. A function that executes on the device is typically called a *kernel*.

3.2.2 A KERNEL CALL

Now we will build upon our example with some code that should look more foreign than our plain-vanilla "Hello, World!" program.

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

This program makes two notable additions to the original "Hello, World!" example:

- An empty function named `kernel()` qualified with `__global__`
- A call to the empty function, embellished with `<<<1,1>>>`

As we saw in the previous section, code is compiled by your system's standard C compiler by default. For example, GNU `gcc` might compile your host code on Linux operating systems, while Microsoft Visual C compiles it on Windows systems. The NVIDIA tools simply feed this host compiler your code, and everything behaves as it would in a world without CUDA.

Now we see that CUDA C adds the `__global__` qualifier to standard C. This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host. In this simple example, `nvcc` gives the function `kernel()` to the compiler that handles device code, and it feeds `main()` to the host compiler as it did in the previous example.

So, what is the mysterious call to `kernel()`, and why must we vandalize our standard C with angle brackets and a numeric tuple? Brace yourself, because this is where the magic happens.

We have seen that CUDA C needed a linguistic method for marking a function as device code. There is nothing special about this; it is shorthand to send host code to one compiler and device code to another compiler. The trick is actually in calling the device code from the host code. One of the benefits of CUDA C is that it provides this language integration so that device function calls look very much like host function calls. Later we will discuss what actually happens behind the scenes, but suffice to say that the CUDA compiler and runtime take care of the messy business of invoking device code from the host.

So, the mysterious-looking call invokes device code, but why the angle brackets and numbers? The angle brackets denote arguments we plan to pass to the runtime system. These are not arguments to the device code but are parameters that will influence how the runtime will launch our device code. We will learn about these parameters to the runtime in the next chapter. Arguments to the device code itself get passed within the parentheses, just like any other function invocation.

3.2.3 PASSING PARAMETERS

We've promised the ability to pass parameters to our kernel, and the time has come for us to make good on that promise. Consider the following enhancement to our "Hello, World!" application:

```
#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;
}
```

You will notice a handful of new lines here, but these changes introduce only two concepts:

- 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.

There is nothing special about passing parameters to a kernel. The angle-bracket syntax notwithstanding, 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.

The more interesting addition is 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, and the second parameter is the size of the allocation you want to make. Besides that your allocated memory pointer is not the function's return value, this is identical behavior to `malloc()`, right down to the `void*` return type. The `HANDLE_ERROR()` that surrounds these calls is a utility macro that we have provided as part of this book's support code. It simply detects that the call has returned an error, prints the associated error message, and exits the application with an `EXIT_FAILURE` code. Although you are free to use this code in your own applications, it is highly likely that this error-handling code will be insufficient in production code.

This raises a subtle but important point. Much of the simplicity and power of CUDA C derives from the ability to blur the line between host and device code. However, 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.

Unfortunately, the compiler cannot protect you from this mistake, either. It will be perfectly happy to allow dereferences of device pointers in your host code because it looks like any other pointer in the application. We can summarize the 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.

If you've been reading carefully, you might have anticipated the next lesson: 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()`, which behaves exactly like `free()` does.

We've seen how to use the host to allocate and free memory on the device, but we've also made it painfully clear that you cannot modify this memory from the host. The remaining two lines of the sample program illustrate two of the most common methods for accessing device memory—by using device pointers from within device code and by using calls to `cudaMemcpy()`.

We use pointers from within device code exactly the same way we use them in standard C that runs on the host code. The statement `*c = a + b` is as simple as it looks. It adds the parameters `a` and `b` together and stores the result in the memory pointed to by `c`. We hope this is almost too easy to even be interesting.

We listed the ways in which we can and cannot use device pointers from within device and host code. These caveats translate exactly as one might imagine when considering host pointers. Although we are free to pass host pointers around in device code, we run into trouble when we attempt to use a host pointer to access memory from within device code. To summarize, host pointers can access memory from host code, and device pointers can access memory from device code.

As promised, 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. In the example, notice that 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.

Unsurprisingly, `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`. If the source and destination pointers are both on the host, we would simply use standard C's `memcpy()` routine to copy between them.

3.3 Querying Devices

Since we would like to be allocating memory and executing code on our device, it would be useful if our program had a way of knowing how much memory and what types of capabilities the device had. Furthermore, it is relatively common for people to have more than one CUDA-capable device per computer. In situations like this, we will definitely want a way to determine which processor is which.

For example, many motherboards ship with integrated NVIDIA graphics processors. When a manufacturer or user adds a discrete graphics processor to this computer, it then possesses two CUDA-capable processors. Some NVIDIA products, like the GeForce GTX 295, ship with two GPUs on a single card. Computers that contain products such as this will also show two CUDA-capable processors.

Before we get too deep into writing device code, we would love to have a mechanism for determining which devices (if any) are present and what capabilities each device supports. Fortunately, there is a very easy interface to determine this information. First, we will want to know how many devices in the system were built on the CUDA Architecture. These devices will be capable of executing kernels written in CUDA C. To get the count of CUDA devices, we call `cudaGetDeviceCount()`. Needless to say, we anticipate receiving an award for Most Creative Function Name.

```
int count;
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

After calling `cudaGetDeviceCount()`, we can then iterate through the devices and query relevant information about each. The CUDA runtime returns us these properties in a structure of type `cudaDeviceProp`. What kind of properties can we retrieve? As of CUDA 3.0, the `cudaDeviceProp` structure contains the following:

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
```

```
        int minor;
        int clockRate;
        size_t textureAlignment;
        int deviceOverlap;
        int multiProcessorCount;
        int kernelExecTimeoutEnabled;
        int integrated;
        int canMapHostMemory;
        int computeMode;
        int maxTexture1D;
        int maxTexture2D[2];
        int maxTexture3D[3];
        int maxTexture2DArray[3];
        int concurrentKernels;
    }
```

Some of these are self-explanatory; others bear some additional description (see Table 3.1).

Table 3.1 CUDA Device Properties

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes

Continued

Table 3.1 Caption needed (Continued)

DEVICE PROPERTY	DESCRIPTION
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
`int computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited
`int maxTexture1D`	The maximum size supported for 1D textures

Table 3.1 CUDA Device Properties (Continued)

DEVICE PROPERTY	DESCRIPTION
`int maxTexture2D[2]`	The maximum dimensions supported for 2D textures
`int maxTexture3D[3]`	The maximum dimensions supported for 3D textures
`int maxTexture2DArray[3]`	The maximum dimensions supported for 2D texture arrays
`int concurrentKernels`	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

We'd like to avoid going too far, too fast down our rabbit hole, so we will not go into extensive detail about these properties now. In fact, the previous list is missing some important details about some of these properties, so you will want to consult the *NVIDIA CUDA Programming Guide* for more information. When you move on to write your own applications, these properties will prove extremely useful. However, for now we will simply show how to query each device and report the properties of each. So far, our device query looks something like this:

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;

    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );

        //Do something with our device's properties

    }
}
```

Now that we know each of the fields available to us, we can expand on the ambiguous "Do something..." section and implement something marginally less trivial:

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;

    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( "   --- General Information for device %d ---\n", i );
        printf( "Name:  %s\n", prop.name );
        printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
        printf( "Clock rate:  %d\n", prop.clockRate );
        printf( "Device copy overlap:  " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
        printf( "Kernel execition timeout :  " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );

        printf( "   --- Memory Information for device %d ---\n", i );
        printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
        printf( "Total constant Mem:  %ld\n", prop.totalConstMem );
        printf( "Max mem pitch:  %ld\n", prop.memPitch );
        printf( "Texture Alignment:  %ld\n", prop.textureAlignment );
        printf( "   --- MP Information for device %d ---\n", i );
        printf( "Multiprocessor count:  %d\n",
                    prop.multiProcessorCount );
        printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
        printf( "Registers per mp:  %d\n", prop.regsPerBlock );
        printf( "Threads in warp:  %d\n", prop.warpSize );
        printf( "Max threads per block:  %d\n",
                    prop.maxThreadsPerBlock );
        printf( "Max thread dimensions:  (%d, %d, %d)\n",
                    prop.maxThreadsDim[0], prop.maxThreadsDim[1],
                    prop.maxThreadsDim[2] );
        printf( "Max grid dimensions:  (%d, %d, %d)\n",
                    prop.maxGridSize[0], prop.maxGridSize[1],
                    prop.maxGridSize[2] );
        printf( "\n" );
    }
}
```

3.4 Using Device Properties

Other than writing an application that handily prints every detail of every CUDA-capable card, why might we be interested in the properties of each device in our system? Since we as software developers want everyone to think our software is fast, we might be interested in choosing the GPU with the most multiprocessors on which to run our code. Or if the kernel needs close interaction with the CPU, we might be interested in running our code on the integrated GPU that shares system memory with the CPU. These are both properties we can query with `cudaGetDeviceProperties()`.

Suppose that we are writing an application that depends on having double-precision floating-point support. After a quick consultation with Appendix A of the *NVIDIA CUDA Programming Guide*, we know that cards that have compute capability 1.3 or higher support double-precision floating-point math. So to successfully run the double-precision application that we've written, we need to find at least one device of compute capability 1.3 or higher.

Based on what we have seen with `cudaGetDeviceCount()` and `cudaGetDeviceProperties()`, we could iterate through each device and look for one that either has a major version greater than 1 or has a major version of 1 and minor version greater than or equal to 3. But since this relatively common procedure is also relatively annoying to perform, the CUDA runtime offers us an automated way to do this. We first fill a `cudaDeviceProp` structure with the properties we need our device to have.

```
    cudaDeviceProp  prop;
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
```

After filling a `cudaDeviceProp` structure, we pass it to `cudaChooseDevice()` to have the CUDA runtime find a device that satisfies this constraint. The call to `cudaChooseDevice()` returns a device ID that we can then pass to `cudaSetDevice()`. From this point forward, all device operations will take place on the device we found in `cudaChooseDevice()`.

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;
    int dev;

    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "ID of current CUDA device:  %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "ID of CUDA device closest to revision 1.3:  %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

Systems with multiple GPUs are becoming more and more common. For example, many of NVIDIA's motherboard chipsets contain integrated, CUDA-capable GPUs. When a discrete GPU is added to one of these systems, you suddenly have a multi-GPU platform. Moreover, NVIDIA's SLI technology allows multiple discrete GPUs to be installed side by side. In either of these cases, your application may have a preference of one GPU over another. If your application depends on certain features of the GPU or depends on having the fastest GPU in the system, you should familiarize yourself with this API because there is no guarantee that the CUDA runtime will choose the best or most appropriate GPU for your application.

3.5 Chapter Review

We've finally gotten our hands dirty writing CUDA C, and ideally it has been less painful than you might have suspected. Fundamentally, CUDA C is standard C with some ornamentation to allow us to specify which code should run on the device and which should run on the host. By adding the keyword `__global__` before a function, we indicated to the compiler that we intend to run the function on the GPU. To use the GPU's dedicated memory, we also learned a CUDA API similar to C's `malloc()`, `memcpy()`, and `free()` APIs. The CUDA versions of these functions, `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()`, allow us to allocate device memory, copy data between the device and host, and free the device memory when we've finished with it.

As we progress through this book, we will see more interesting examples of how we can effectively use the device as a massively parallel coprocessor. For now, you should know how easy it is to get started with CUDA C, and in the next chapter we will see how easy it is to execute parallel code on the GPU.

Chapter 4

Parallel Programming in CUDA C

In the previous chapter, we saw how simple it can be to write code that executes on the GPU. We have even gone so far as to learn how to add two numbers together, albeit just the numbers 2 and 7. Admittedly, that example was not immensely impressive, nor was it incredibly interesting. But we hope you are convinced that it is easy to get started with CUDA C and you're excited to learn more. Much of the promise of GPU computing lies in exploiting the massively parallel structure of many problems. In this vein, we intend to spend this chapter examining how to execute parallel code on the GPU using CUDA C.

.1 Chapter Objectives

Through the course of this chapter, you will accomplish the following:

- You will learn one of the fundamental ways CUDA exposes its parallelism.
- You will write your first parallel code with CUDA C.

.2 CUDA Parallel Programming

Previously, we saw how easy it was to get a standard C function to start running on a device. By adding the `__global__` qualifier to the function and by calling it using a special angle bracket syntax, we executed the function on our GPU. Although this was extremely simple, it was also extremely inefficient because NVIDIA's hardware engineering minions have optimized their graphics processors to perform hundreds of computations in parallel. However, thus far we have only ever launched a kernel that runs serially on the GPU. In this chapter, we see how straightforward it is to launch a device kernel that performs its computations in parallel.

4.2.1 SUMMING VECTORS

We will contrive a simple example to illustrate threads and how we use them to code with CUDA C. Imagine having two lists of numbers where we want to sum corresponding elements of each list and store the result in a third list. Figure 4.1 shows this process. If you have any background in linear algebra, you will recognize this operation as summing two vectors.

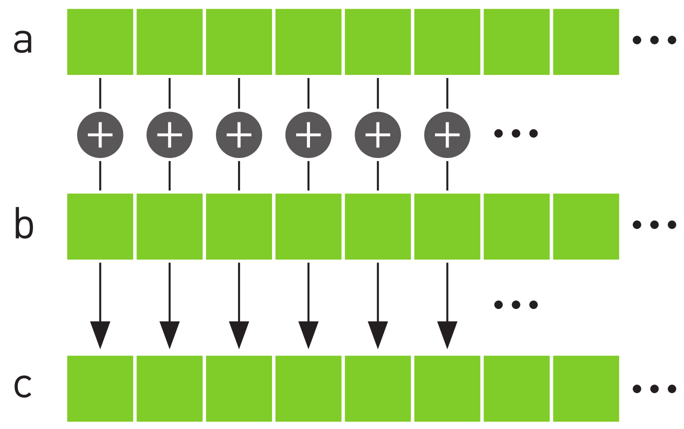


Figure 4.1 Summing two vectors

CPU VECTOR SUMS

First we'll look at one way this addition can be accomplished with traditional C code:

```
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Most of this example bears almost no explanation, but we will briefly look at the `add()` function to explain why we overly complicated it.

```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
```

We compute the sum within a `while` loop where the index `tid` ranges from `0` to `N-1`. We add corresponding elements of `a[]` and `b[]`, placing the result in the corresponding element of `c[]`. One would typically code this in a slightly simpler manner, like so:

```
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Our slightly more convoluted method was intended to suggest a potential way to parallelize the code on a system with multiple CPUs or CPU cores. For example, with a dual-core processor, one could change the increment to 2 and have one core initialize the loop with `tid = 0` and another with `tid = 1`. The first core would add the even-indexed elements, and the second core would add the odd-indexed elements. This amounts to executing the following code on each of the two CPU cores:

CPU CORE 1	**CPU CORE 2**

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```
	```
	void add( int *a, int *b, int *c )
	{
	    int tid = 1;
	    while (tid < N) {
	        c[tid] = a[tid] + b[tid];
	        tid += 2;
	    }
	}
	```

Of course, doing this on a CPU would require considerably more code than we have included in this example. You would need to provide a reasonable amount of infrastructure to create the worker threads that execute the function `add()` as well as make the assumption that each thread would execute in parallel, a scheduling assumption that is unfortunately not always true.

GPU VECTOR SUMS

We can accomplish the same addition very similarly on a GPU by writing `add()` as a device function. This should look similar to code you saw in the previous chapter. But before we look at the device code, we present `main()`. Although the GPU implementation of `main()` is different from the corresponding CPU version, nothing here should look new:

```
#include "../common/book.h"

#define N   10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

You will notice some common patterns that we employ again:

- We allocate three arrays on the device using calls to `cudaMalloc()`: two arrays, `dev_a` and `dev_b`, to hold inputs, and one array, `dev_c`, to hold the result.
- Because we are environmentally conscientious coders, we clean up after ourselves with `cudaFree()`.
- Using `cudaMemcpy()`, we copy the input data to the device with the parameter `cudaMemcpyHostToDevice` and copy the result data back to the host with `cudaMemcpyDeviceToHost`.
- We execute the device code in `add()` from the host code in `main()` using the triple angle bracket syntax.

As an aside, you may be wondering why we fill the input arrays on the CPU. There is no reason in particular why we *need* to do this. In fact, the performance of this step would be faster if we filled the arrays on the GPU. But we intend to show how a particular operation, namely, the addition of two vectors, can be implemented on a graphics processor. As a result, we ask you to imagine that this is but one step of a larger application where the input arrays `a[]` and `b[]` have been generated by some other algorithm or loaded from the hard drive by the user. In summary, it will suffice to pretend that this data appeared out of nowhere and now we need to do something with it.

Moving on, our `add()` routine looks similar to its corresponding CPU implementation:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;     // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Again we see a common pattern with the function `add()`:

- We have written a function called `add()` that executes on the device. We accomplished this by taking C code and adding a `__global__` qualifier to the function name.

So far, there is nothing new in this example except it can do more than add 2 and 7. However, there *are* two noteworthy components of this example: The parameters within the triple angle brackets and the code contained in the kernel itself both introduce new concepts.

Up to this point, we have always seen kernels launched in the following form:

```
        kernel<<<1,1>>>( param1, param2, … );
```

But in this example we are launching with a number in the angle brackets that is not 1:

```
        add<<<N,1>>>( dev_a, dev_b, dev_c );
```

What gives?

Recall that we left those two numbers in the angle brackets unexplained; we stated vaguely that they were parameters to the runtime that describe how to launch the kernel. Well, the first number in those parameters represents the number of parallel blocks in which we would like the device to execute our kernel. In this case, we're passing the value `N` for this parameter.

For example, if we launch with `kernel<<<2,1>>>()`, you can think of the runtime creating two copies of the kernel and running them in parallel. We call each of these parallel invocations a *block*. With `kernel<<<256,1>>>()`, you would get 256 *blocks* running on the GPU. Parallel programming has never been easier.

But this raises an excellent question: The GPU runs `N` copies of our kernel code, but how can we tell from within the code which block is currently running? This question brings us to the second new feature of the example, the kernel code itself. Specifically, it brings us to the variable `blockIdx.x`:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

At first glance, it looks like this variable should cause a syntax error at compile time since we use it to assign the value of `tid`, but we have never defined it. However, there is no need to define the variable `blockIdx`; this is one of the built-in variables that the CUDA runtime defines for us. Furthermore, we use this variable for exactly what it sounds like it means. It contains the value of the block index for whichever block is currently running the device code.

Why, you may then ask, is it not just `blockIdx`? Why `blockIdx.x`? As it turns out, CUDA C allows you to define a group of blocks in two dimensions. For problems with two-dimensional domains, such as matrix math or image processing, it is often convenient to use two-dimensional indexing to avoid annoying translations from linear to rectangular indices. Don't worry if you aren't familiar with these problem types; just know that using two-dimensional indexing can sometimes be more convenient than one-dimensional indexing. But you never *have* to use it. We won't be offended.

When we launched the kernel, we specified N as the number of parallel blocks. We call the collection of parallel blocks a *grid*. This specifies to the runtime system that we want a one-dimensional *grid* of N blocks (scalar values are interpreted as one-dimensional). These threads will have varying values for blockIdx.x, the first taking value 0 and the last taking value N-1. So, imagine four blocks, all running through the same copy of the device code but having different values for the variable blockIdx.x. This is what the actual code being executed in each of the four parallel blocks looks like after the runtime substitutes the appropriate block index for blockIdx.x:

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

If you recall the CPU-based example with which we began, you will recall that we needed to walk through indices from 0 to N-1 in order to sum the two vectors. Since the runtime system is already launching a kernel where each block will have one of these indices, nearly all of this work has already been done for us. Because we're something of a lazy lot, this is a good thing. It affords us more time to blog, probably about how lazy we are.

The last remaining question to be answered is, why do we check whether tid is less than N? It *should* always be less than N, since we've specifically launched our kernel such that this assumption holds. But our desire to be lazy also makes us paranoid about someone breaking an assumption we've made in our code. Breaking code assumptions means broken code. This means bug reports, late nights tracking down bad behavior, and generally lots of activities that stand between us and our blog. If we didn't check that `tid` is less than `N` and subsequently fetched memory that wasn't ours, this would be bad. In fact, it could possibly kill the execution of your kernel, since GPUs have sophisticated memory management units that kill processes that seem to be violating memory rules.

If you encounter problems like the ones just mentioned, one of the `HANDLE_ERROR()` macros that we've sprinkled so liberally throughout the code will detect and alert you to the situation. As with traditional C programming, the lesson here is that functions return error codes for a reason. Although it is always tempting to ignore these error codes, we would love to save *you* the hours of pain through which *we* have suffered by urging that you *check the results of every operation that can fail*. As is often the case, the presence of these errors will not prevent you from continuing the execution of your application, but they will most certainly cause all manner of unpredictable and unsavory side effects downstream.

At this point, you're running code in parallel on the GPU. Perhaps you had heard this was tricky or that you had to understand computer graphics to do general-purpose programming on a graphics processor. We hope you are starting to see how CUDA C makes it much easier to get started writing parallel code on a GPU. We used the example only to sum vectors of length 10. If you would like to see how easy it is to generate a massively parallel application, try changing the 10 in the line `#define N 10` to 10000 or 50000 to launch tens of thousands of parallel blocks. Be warned, though: No dimension of your launch of blocks may exceed 65,535. This is simply a hardware-imposed limit, so you will start to see failures if you attempt launches with more blocks than this. In the next chapter, we will see how to work within this limitation.

4.2.2 A FUN EXAMPLE

We don't mean to imply that adding vectors is anything less than fun, but the following example will satisfy those looking for some flashy examples of parallel CUDA C.

The following example will demonstrate code to draw slices of the Julia Set. For the uninitiated, the Julia Set is the boundary of a certain class of functions over complex numbers. Undoubtedly, this sounds even less fun than vector addition and matrix multiplication. However, for almost all values of the function's parameters, this boundary forms a fractal, one of the most interesting and beautiful curiosities of mathematics.

The calculations involved in generating such a set are quite simple. At its heart, the Julia Set evaluates a simple iterative equation for points in the complex plane. A point is *not* in the set if the process of iterating the equation diverges for that point. That is, if the sequence of values produced by iterating the equation grows toward infinity, a point is considered *outside* the set. Conversely, if the values taken by the equation remain bounded, the point *is* in the set.

Computationally, the iterative equation in question is remarkably simple, as shown in Equation 4.1.

Equation 4.1

$$Z_{n+1} = Z_n^2 + C$$

Computing an iteration of Equation 4.1 would therefore involve squaring the current value and adding a constant to get the next value of the equation.

CPU JULIA SET

We will examine a source listing now that will compute and visualize the Julia Set. Since this is a more complicated program than we have studied so far, we will split it into pieces here. Later in the chapter, you will see the entire source listing.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

Our main routine is remarkably simple. It creates the appropriate size bitmap image using a utility library provided. Next, it passes a pointer to the bitmap data to the kernel function.

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
 }
```

The computation kernel does nothing more than iterate through all points we care to render, calling `julia()` on each to determine membership in the Julia Set. The function `julia()` will return 1 if the point is in the set and 0 if it is not in the set. We set the point's color to be red if `julia()` returns 1 and black if it returns 0. These colors are arbitrary, and you should feel free to choose a color scheme that matches your personal aesthetics.

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

This function is the meat of the example. We begin by translating our pixel coordinate to a coordinate in complex space. To center the complex plane at the image center, we shift by `DIM/2`. Then, to ensure that the image spans the range of -1.0 to 1.0, we scale the image coordinate by `DIM/2`. Thus, given an image point at `(x,y)`, we get a point in complex space at `( (DIM/2 – x)/(DIM/2), ((DIM/2 – y)/(DIM/2) )`.

Then, to potentially zoom in or out, we introduce a `scale` factor. Currently, the scale is hard-coded to be 1.5, but you should tweak this parameter to zoom in or out. If you are feeling really ambitious, you could make this a command-line parameter.

After obtaining the point in complex space, we then need to determine whether the point is in or out of the Julia Set. If you recall the previous section, we do this by computing the values of the iterative equation $Z_{n+1} = z_n^2 + C$. Since C is some arbitrary complex-valued constant, we have chosen `-0.8 + 0.156i` because it happens to yield an interesting picture. You should play with this constant if you want to see other versions of the Julia Set.

In the example, we compute 200 iterations of this function. After each iteration, we check whether the magnitude of the result exceeds some threshold (1,000 for our purposes). If so, the equation is diverging, and we can return 0 to indicate that the point is *not* in the set. On the other hand, if we finish all 200 iterations and the magnitude is still bounded under 1,000, we assume that the point is in the set, and we return 1 to the caller, `kernel()`.

Since all the computations are being performed on complex numbers, we define a generic structure to store complex numbers.

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

The class represents complex numbers with two data elements: a single-precision real component `r` and a single-precision imaginary component `i`. The class defines addition and multiplication operators that combine complex numbers as expected. (If you are completely unfamiliar with complex numbers, you can get a quick primer online.) Finally, we define a method that returns the magnitude of the complex number.

GPU JULIA SET

The device implementation is remarkably similar to the CPU version, continuing a trend you may have noticed.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                              dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    cudaFree( dev_bitmap );
}
```

This version of `main()` looks much more complicated than the CPU version, but the flow is actually identical. Like with the CPU version, we create a `DIM` x `DIM` bitmap image using our utility library. But because we will be doing computation on a GPU, we also declare a pointer called `dev_bitmap` to hold a copy of the data on the device. And to hold data, we need to allocate memory using `cudaMalloc()`.

We then run our `kernel()` function exactly like in the CPU version, although now it is a `__global__` function, meaning it will run on the GPU. As with the CPU example, we pass `kernel()` the pointer we allocated in the previous line to store the results. The only difference is that the memory resides on the GPU now, not on the host system.

The most significant difference is that we specify how many parallel blocks on which to execute the function `kernel()`. Because each point can be computed independently of every other point, we simply specify one copy of the function for each point we want to compute. We mentioned that for some problem domains, it helps to use two-dimensional indexing. Unsurprisingly, computing function values over a two-dimensional domain such as the complex plane is one of these problems. So, we specify a two-dimensional grid of blocks in this line:

```
    dim3 grid(DIM,DIM);
```

The type `dim3` is not a standard C type, lest you feared you had forgotten some key pieces of information. Rather, the CUDA runtime header files define some convenience types to encapsulate multidimensional tuples. The type `dim3` represents a three-dimensional tuple that will be used to specify the size of our launch. But why do we use a three-dimensional value when we oh-so-clearly stated that our launch is a *two-dimensional* grid?

Frankly, we do this because a three-dimensional, `dim3` value is what the CUDA runtime expects. Although a three-dimensional launch grid is not currently supported, the CUDA runtime still expects a `dim3` variable where the last component equals 1. When we initialize it with only two values, as we do in the statement `dim3 grid(DIM,DIM)`, the CUDA runtime automatically fills the third dimension with the value 1, so everything here will work as expected. Although it's possible that NVIDIA will support a three-dimensional grid in the future, for now we'll just play nicely with the kernel launch API because when coders and APIs fight, the API always wins.

We then pass our `dim3` variable `grid` to the CUDA runtime in this line:

```
kernel<<<grid,1>>>( dev _ bitmap );
```

Finally, a consequence of the results residing on the device is that after executing `kernel()`, we have to copy the results back to the host. As we learned in previous chapters, we accomplish this with a call to `cudaMemcpy()`, specifying the direction `cudaMemcpyDeviceToHost` as the last argument.

```
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                              dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
```

One of the last wrinkles in the difference of implementation comes in the implementation of `kernel()`.

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

First, we need `kernel()` to be declared as a `__global__` function so it runs on the device but can be called from the host. Unlike the CPU version, we no longer need nested `for()` loops to generate the pixel indices that get passed to `julia()`. As with the vector addition example, the CUDA runtime generates these indices for us in the variable `blockIdx`. This works because we declared our grid of blocks to have the same dimensions as our image, so we get one block for each pair of integers `(x,y)` between `(0,0)` and `(DIM-1, DIM-1)`.

Next, the only additional information we need is a linear offset into our output buffer, `ptr`. This gets computed using another built-in variable, `gridDim`. This variable is a constant across all blocks and simply holds the dimensions of the grid that was launched. In this example, it will always be the value `(DIM, DIM)`. So, multiplying the row index by the grid width and adding the column index will give us a unique index into `ptr` that ranges from `0` to `(DIM*DIM-1)`.

```
    int offset = x + y * gridDim.x;
```

Finally, we examine the actual code that determines whether a point is in or out of the Julia Set. This code should look identical to the CPU version, continuing a trend we have seen in many examples now.

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Again, we define a `cuComplex` structure that defines a method for storing a complex number with single-precision floating-point components. The structure also defines addition and multiplication operators as well as a function to return the magnitude of the complex value.

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Notice that we use the same language constructs in CUDA C that we use in our CPU version. The one difference is the qualifier `__device__`, which indicates that this code will run on a GPU and not on the host. Recall that because these functions are declared as `__device__` functions, they will be callable only from other `__device__` functions or from `__global__` functions.

Since we've interrupted the code with commentary so frequently, here is the entire source listing from start to finish:

```
#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 1000
```

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};

__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    HANDLE_ERROR( cudaFree( dev_bitmap ) );
}
```

When you run the application, you should see an animating visualization of the Julia Set. To convince you that it has earned the title "A Fun Example," Figure 4.2 shows a screenshot taken from this application.

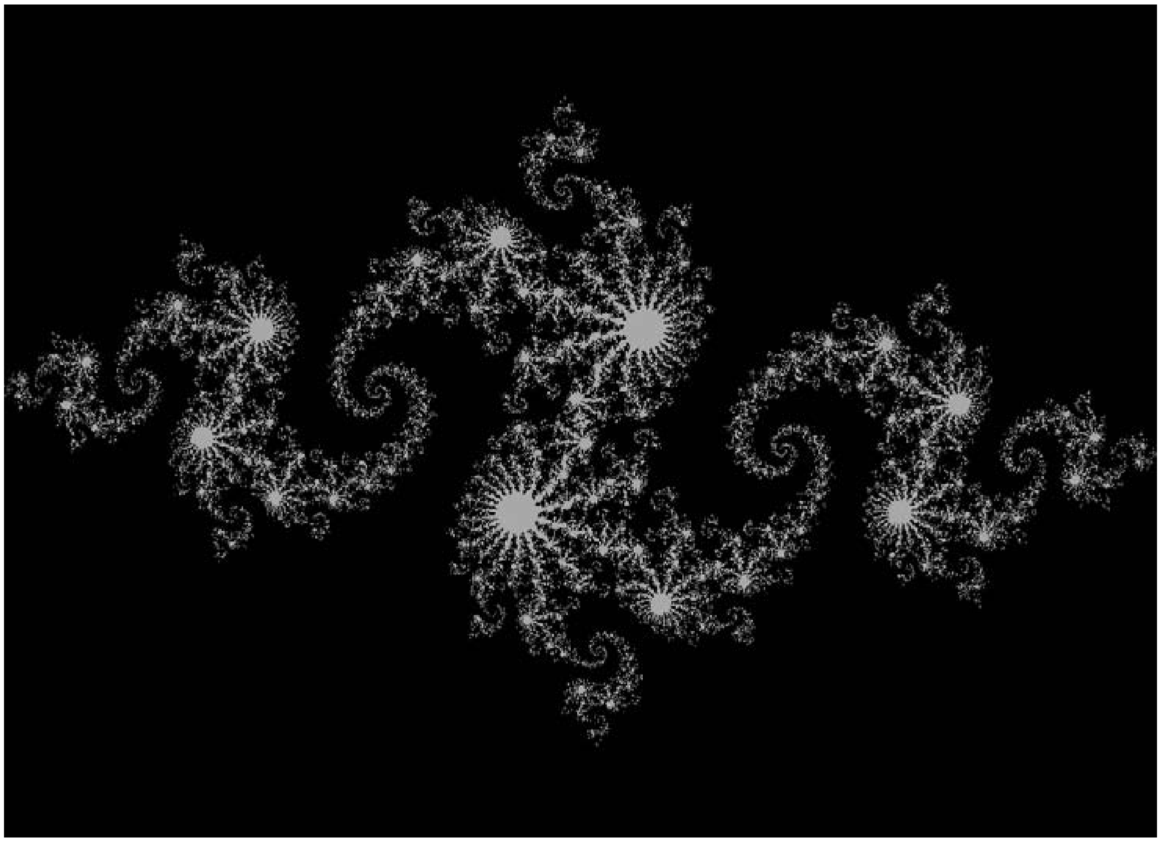


Figure 4.2 A screenshot from the GPU Julia Set application

4.3 Chapter Review

Congratulations, you can now write, compile, and run massively parallel code on a graphics processor! You should go brag to your friends. And if they are still under the misconception that GPU computing is exotic and difficult to master, they will be most impressed. The ease with which you accomplished it will be our secret. If they're people you trust with your secrets, suggest that they buy the book, too.

We have so far looked at how to instruct the CUDA runtime to execute multiple copies of our program in parallel on what we called *blocks*. We called the collection of blocks we launch on the GPU a *grid*. As the name might imply, a grid can be either a one- or two-dimensional collection of blocks. Each copy of the kernel can determine which block it is executing with the built-in variable `blockIdx`. Likewise, it can determine the size of the grid by using the built-in variable `gridDim`. Both of these built-in variables proved useful within our kernel to calculate the data index for which each block is responsible.

Chapter 5

Thread Cooperation

We have now written our first program using CUDA C as well as have seen how to write code that executes in parallel on a GPU. This is an excellent start! But arguably one of the most important components to parallel programming is the means by which the parallel processing elements cooperate on solving a problem. Rare are the problems where every processor can compute results and terminate execution without a passing thought as to what the other processors are doing. For even moderately sophisticated algorithms, we will need the parallel copies of our code to communicate and cooperate. So far, we have not seen any mechanisms for accomplishing this communication between sections of CUDA C code executing in parallel. Fortunately, there is a solution, one that we will begin to explore in this chapter.

.1 Chapter Objectives

Through the course of this chapter, you will accomplish the following:

- You will learn about what CUDA C calls *threads*.
- You will learn a mechanism for different threads to communicate with each other.
- You will learn a mechanism to synchronize the parallel execution of different threads.

.2 Splitting Parallel Blocks

In the previous chapter, we looked at how to launch parallel code on the GPU. We did this by instructing the CUDA runtime system on how many parallel copies of our kernel to launch. We call these parallel copies *blocks*.

The CUDA runtime allows these blocks to be split into *threads*. Recall that when we launched multiple parallel blocks, we changed the first argument in the angle brackets from 1 to the number of blocks we wanted to launch. For example, when we studied vector addition, we launched a block for each element in the vector of size N by calling this:

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

Inside the angle brackets, the second parameter actually represents the number of threads per block we want the CUDA runtime to create on our behalf. To this point, we have only ever launched one thread per block. In the previous example, we launched the following:

N blocks x 1 thread/block = N parallel threads

So really, we could have launched `N/2` blocks with two threads per block, `N/4` blocks with four threads per block, and so on. Let's revisit our vector addition example armed with this new information about the capabilities of CUDA C.

5.2.1 VECTOR SUMS: REDUX

We endeavor to accomplish the same task as we did in the previous chapter. That is, we want to take two input vectors and store their sum in a third output vector. However, this time we will use threads instead of blocks to accomplish this.

You may be wondering, what is the advantage of using threads rather than blocks? Well, for now, there is no advantage worth discussing. But parallel threads within a block will have the ability to do things that parallel blocks cannot do. So for now, be patient and humor us while we walk through a parallel thread version of the parallel block example from the previous chapter.

GPU VECTOR SUMS USING THREADS

We will start by addressing the two changes of note when moving from parallel blocks to parallel threads. Our kernel invocation will change from one that launches `N` blocks of one thread apiece:

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

to a version that launches `N` threads, all within one block:

```
add<<<1,N>>>( dev_a, dev_b, dev_c );
```

The only other change arises in the method by which we index our data. Previously, within our kernel we indexed the input and output data by block index.

```
int tid = blockIdx.x;
```

The punch line here should not be a surprise. Now that we have only a single block, we have to index the data by thread index.

```
int tid = threadIdx.x;
```

These are the only two changes required to move from a parallel block implementation to a parallel thread implementation. For completeness, here is the entire source listing with the changed lines in bold:

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

   // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a,
                              a,
                              N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b,
                              b,
                              N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<1,N>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c,
                              dev_c,
                              N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
```

```
        // free the memory allocated on the GPU
        cudaFree( dev_a );
        cudaFree( dev_b );
        cudaFree( dev_c );

        return 0;
    }
```

Pretty simple stuff, right? In the next section, we'll see one of the limitations of this thread-only approach. And of course, later we'll see why we would even bother splitting blocks into other parallel components.

GPU SUMS OF A LONGER VECTOR

In the previous chapter, we noted that the hardware limits the number of blocks in a single launch to 65,535. Similarly, the hardware limits the number of threads per block with which we can launch a kernel. Specifically, this number cannot exceed the value specified by the `maxThreadsPerBlock` field of the device properties structure we looked at in Chapter 3. For many of the graphics processors currently available, this limit is 512 threads per block, so how would we use a thread-based approach to add two vectors of size greater than 512? We will have to use a combination of threads and blocks to accomplish this.

As before, this will require two changes: We will have to change the index computation within the kernel, and we will have to change the kernel launch itself.

Now that we have multiple blocks and threads, the indexing will start to look similar to the standard method for converting from a two-dimensional index space to a linear space.

```
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

This assignment uses a new built-in variable, `blockDim`. This variable is a constant for all blocks and stores the number of threads along each dimension of the block. Since we are using a one-dimensional block, we refer only to `blockDim.x`. If you recall, `gridDim` stored a similar value, but it stored the number of blocks along each dimension of the entire grid. Moreover, `gridDim` is two-dimensional, whereas `blockDim` is actually three-dimensional. That is, the CUDA runtime allows you to launch a two-dimensional grid of blocks where each block is a three-dimensional array of threads. Yes, this is a lot of dimensions, and it is unlikely you will regularly need the five degrees of indexing freedom afforded you, but they are available if so desired.

Indexing the data in a linear array using the previous assignment actually is quite intuitive. If you disagree, it may help to think about your collection of blocks of threads spatially, similar to a two-dimensional array of pixels. We depict this arrangement in Figure 5.1.

If the threads represent columns and the blocks represent rows, we can get a unique index by taking the product of the block index with the number of threads in each block and adding the thread index within the block. This is identical to the method we used to linearize the two-dimensional image index in the Julia Set example.

```
    int offset = x + y * DIM;
```

Here, `DIM` is the block dimension (measured in threads), `y` is the block index, and `x` is the thread index within the block. Hence, we arrive at the index: `tid = threadIdx.x + blockIdx.x * blockDim.x`.

The other change is to the kernel launch itself. We still need `N` parallel threads to launch, but we want them to launch across multiple blocks so we do not hit the 512-thread limitation imposed upon us. One solution is to arbitrarily set the block size to some fixed number of threads; for this example, let's use 128 threads per block. Then we can just launch `N/128` blocks to get our total of `N` threads running.

The wrinkle here is that `N/128` is an integer division. This implies that if `N` were 127, `N/128` would be zero, and we will not actually compute anything if we launch

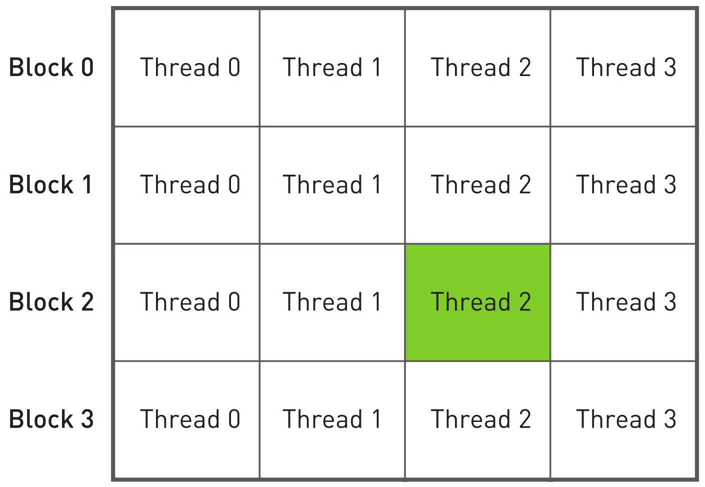


Figure 5.1 A two-dimensional arrangement of a collection of blocks and threads

zero threads. In fact, we will launch too few threads whenever N is not an exact multiple of 128. This is bad. We actually want this division to round up.

There is a common trick to accomplish this in integer division without calling ceil(). We actually compute (N+127)/128 instead of N/128. Either you can take our word that this will compute the smallest multiple of 128 greater than or equal to N or you can take a moment now to convince yourself of this fact.

We have chosen 128 threads per block and therefore use the following kernel launch:

```
    add<<< (N+127)/128, 128 >>>( dev _ a, dev _ b, dev _ c );
```

Because of our change to the division that ensures we launch enough threads, we will actually now launch *too many* threads when N is not an exact multiple of 128. But there is a simple remedy to this problem, and our kernel already takes care of it. We have to check whether a thread's offset is actually between 0 and N before we use it to access our input and output arrays:

```
    if (tid < N)
        c[tid] = a[tid] + b[tid];
```

Thus, when our index overshoots the end of our array, as will always happen when we launch a nonmultiple of 128, we automatically refrain from performing the calculation. More important, we refrain from reading and writing memory off the end of our array.

GPU SUMS OF ARBITRARILY LONG VECTORS

We were not completely forthcoming when we first discussed launching parallel blocks on a GPU. In addition to the limitation on thread count, there is also a hardware limitation on the number of blocks (albeit much greater than the thread limitation). As we've mentioned previously, neither dimension of a grid of blocks may exceed 65,535.

So, this raises a problem with our current vector addition implementation. If we launch N/128 blocks to add our vectors, we will hit launch failures when our vectors exceed 65,535 * 128 = 8,388,480 elements. This seems like a large number, but with current memory capacities between 1GB and 4GB, the high-end graphics processors can hold orders of magnitude more data than vectors with 8 million elements.

Fortunately, the solution to this issue is extremely simple. We first make a change to our kernel.

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

This looks remarkably like our *original* version of vector addition! In fact, compare it to the following CPU implementation from the previous chapter:

```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
```

Here we also used a `while()` loop to iterate through the data. Recall that we claimed that rather than incrementing the array index by 1, a multi-CPU or multi-core version could increment by the number of processors we wanted to use. We will now use that same principle in the GPU version.

In the GPU implementation, we consider the number of parallel threads launched to be the number of processors. Although the actual GPU may have fewer (or more) processing units than this, we think of each thread as logically executing in parallel and then allow the hardware to schedule the actual execution. Decoupling the parallelization from the actual method of hardware execution is one of burdens that CUDA C lifts off a software developer's shoulders. This should come as a relief, considering current NVIDIA hardware can ship with anywhere between 8 and 480 arithmetic units per chip!

Now that we understand the principle behind this implementation, we just need to understand how we determine the initial index value for each parallel thread and how we determine the increment. We want each parallel thread to start on a different data index, so we just need to take our thread and block indexes and linearize them as we saw in the "GPU Sums of a Longer Vector" section. Each thread will start at an index given by the following:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

After each thread finishes its work at the current index, we need to increment each of them by the total number of threads running in the grid. This is simply the number of threads per block multiplied by the number of blocks in the grid, or `blockDim.x * gridDim.x`. Hence, the increment step is as follows:

```
tid += blockDim.x * gridDim.x;
```

We are almost there! The only remaining piece is to fix the launch itself. If you remember, we took this detour because the launch `add<<<(N+127)/128,128>>>( dev_a, dev_b, dev_c )` will fail when `(N+127)/128` is greater than 65,535. To ensure we never launch too many blocks, we will just fix the number of blocks to some reasonably small value. Since we like copying and pasting so much, we will use 128 blocks, each with 128 threads.

```
add<<<128,128>>>( dev _ a, dev _ b, dev _ c );
```

You should feel free to adjust these values however you see fit, provided that your values remain within the limits we've discussed. Later in the book, we will discuss the potential performance implications of these choices, but for now it suffices to choose 128 threads per block and 128 blocks. Now we can add vectors of arbitrary length, limited only by the amount of RAM we have on our GPU. Here is the entire source listing:

```
#include "../common/book.h"

#define N (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

```
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a,
                              a,
                              N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b,
                              b,
                              N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<128,128>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c,
                              dev_c,
                              N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
            }
        }
        if (success)    printf( "We did it!\n" );

        // free the memory allocated on the GPU
        cudaFree( dev_a );
        cudaFree( dev_b );
        cudaFree( dev_c );

        return 0;
    }
```

5.2.2 GPU RIPPLE USING THREADS

As with the previous chapter, we will reward your patience with vector addition by presenting a more fun example that demonstrates some of the techniques we've been using. We will again use our GPU computing power to generate pictures procedurally. But to make things even more interesting, this time we will animate them. But don't worry, we've packaged all the unrelated animation code into helper functions so you won't have to master any graphics or animation.

```
struct DataBlock {
    unsigned char   *dev_bitmap;
    CPUAnimBitmap  *bitmap;
};

// clean up memory allocated on the GPU
void cleanup( DataBlock *d ) {
    cudaFree( d->dev_bitmap );
}

int main( void ) {
    DataBlock   data;
    CPUAnimBitmap  bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_bitmap,
                              bitmap.image_size() ) );

    bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
                          (void (*)(void*))cleanup );
}
```

Most of the complexity of `main()` is hidden in the helper class `CPUAnimBitmap`. You will notice that we again have a pattern of doing a `cudaMalloc()`, executing device code that uses the allocated memory, and then cleaning up with `cudaFree()`. This should be old hat to you by now.

In this example, we have slightly convoluted the means by which we accomplish the middle step, "executing device code that uses the allocated memory." We pass the `anim_and_exit()` method a function pointer to `generate_frame()`. This function will be called by the class every time it wants to generate a new frame of the animation.

```
void generate_frame( DataBlock *d, int ticks ) {
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<blocks,threads>>>( d->dev_bitmap, ticks );

    HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr(),
                              d->dev_bitmap,
                              d->bitmap->image_size(),
                              cudaMemcpyDeviceToHost ) );
}
```

Although this function consists only of four lines, they all involve important CUDA C concepts. First, we declare two two-dimensional variables, `blocks` and `threads`. As our naming convention makes painfully obvious, the variable `blocks` represents the number of parallel blocks we will launch in our grid. The variable `threads` represents the number of threads we will launch per block. Because we are generating an image, we use two-dimensional indexing so that each thread will have a unique `(x,y)` index that we can easily put into correspondence with a pixel in the output image. We have chosen to use blocks that consist of a 16 x 16 array of threads. If the image has `DIM` x `DIM` pixels, we need to launch `DIM/16` x `DIM/16` blocks to get one thread per pixel. Figure 5.2 shows how this block and thread configuration would look in a (ridiculously) small, 48-pixel-wide, 32-pixel-high image.

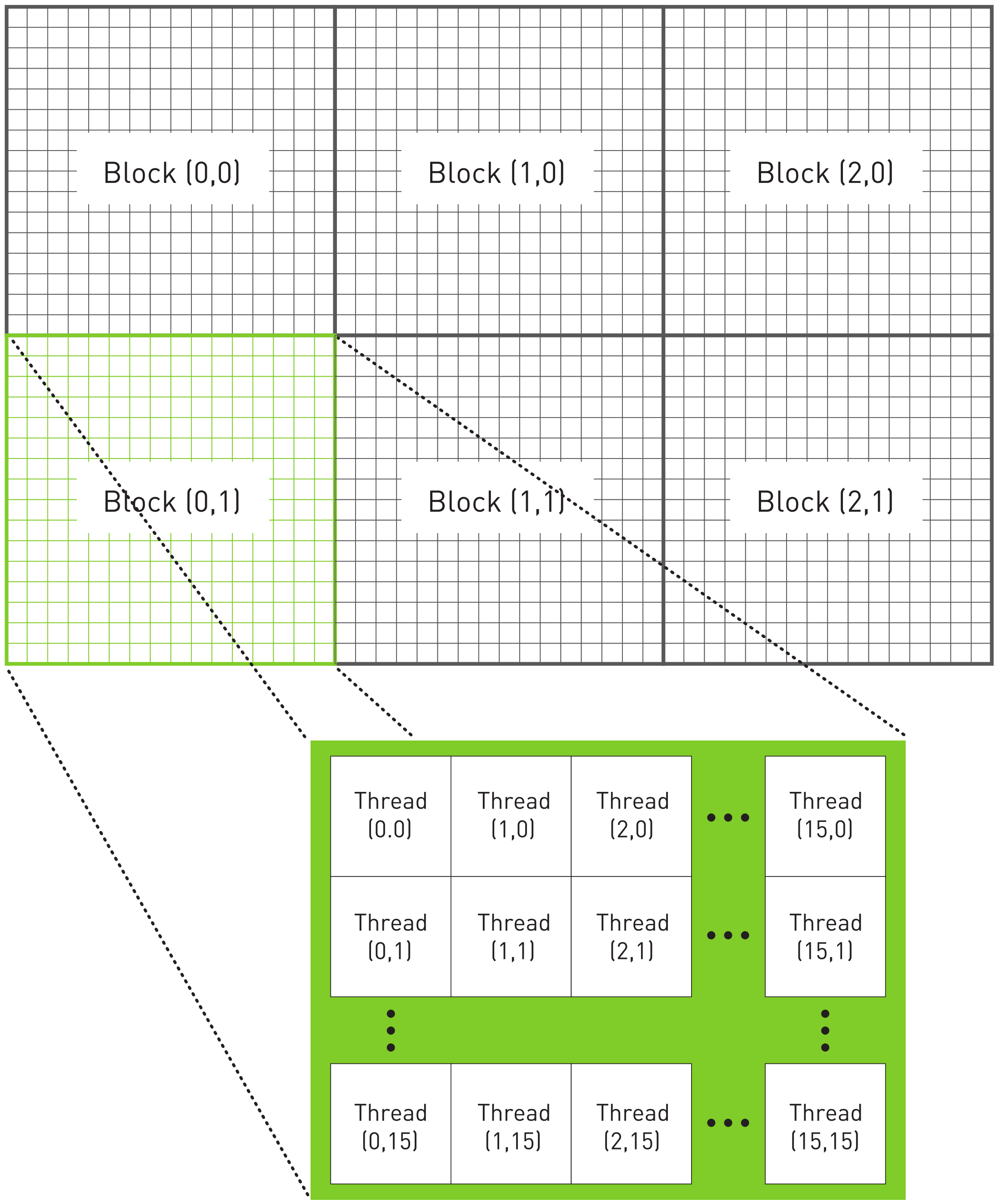


Figure 5.2 A 2D hierarchy of blocks and threads that could be used to process a 48 x 32 pixel image using one thread per pixel

If you have done any multithreaded CPU programming, you may be wondering why we would launch so many threads. For example, to render a full high-definition animation at 1920 x 1080, this method would create more than 2 million threads. Although we routinely create and schedule this many threads on a GPU, one would not dream of creating this many threads on a CPU. Because CPU thread management and scheduling must be done in software, it simply cannot scale to the number of threads that a GPU can. Because we can simply create a thread for each data element we want to process, parallel programming on a GPU can be far simpler than on a CPU.

After declaring the variables that hold the dimensions of our launch, we simply launch the kernel that will compute our pixel values.

```
kernel<<< blocks,threads>>>( d->dev _ bitmap, ticks );
```

The kernel will need two pieces of information that we pass as parameters. First, it needs a pointer to device memory that holds the output pixels. This is a global variable that had its memory allocated in `main()`. But the variable is “global” only for host code, so we need to pass it as a parameter to ensure that the CUDA runtime will make it available for our device code.

Second, our kernel will need to know the current animation time so it can generate the correct frame. The current time, `ticks`, is passed to the `generate_frame()` function from the infrastructure code in `CPUAnimBitmap`, so we can simply pass this on to our kernel.

And now, here’s the kernel code itself:

```
__global__ void kernel( unsigned char *ptr, int ticks ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // now calculate the value at that position
    float fx = x - DIM/2;
    float fy = y - DIM/2;
    float d = sqrtf( fx * fx + fy * fy );
```

```
        unsigned char grey = (unsigned char)(128.0f + 127.0f *
                                             cos(d/10.0f - ticks/7.0f) /
                                             (d/10.0f + 1.0f));
        ptr[offset*4 + 0] = grey;
        ptr[offset*4 + 1] = grey;
        ptr[offset*4 + 2] = grey;
        ptr[offset*4 + 3] = 255;
}
```

The first three are the most important lines in the kernel.

```
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
```

In these lines, each thread takes its index within its block as well as the index of its block within the grid, and it translates this into a unique `(x,y)` index within the image. So when the thread at index `(3, 5)` in block `(12, 8)` begins executing, it knows that there are 12 entire blocks to the left of it and 8 entire blocks above it. Within its block, the thread at `(3, 5)` has three threads to the left and five above it. Because there are 16 threads per block, this means the thread in question has the following:

3 threads + 12 blocks * 16 threads/block = 195 threads to the left of it

5 threads + 8 blocks * 16 threads/block = 128 threads above it

This computation is identical to the computation of `x` and `y` in the first two lines and is how we map the thread and block indices to image coordinates. Then we simply linearize these `x` and `y` values to get an offset into the output buffer. Again, this is identical to what we did in the "GPU Sums of a Longer Vector" and "GPU Sums of Arbitrarily Long Vectors" sections.

```
int offset = x + y * blockDim.x * gridDim.x;
```

Since we know which `(x,y)` pixel in the image the thread should compute and we know the time at which it needs to compute this value, we can compute any