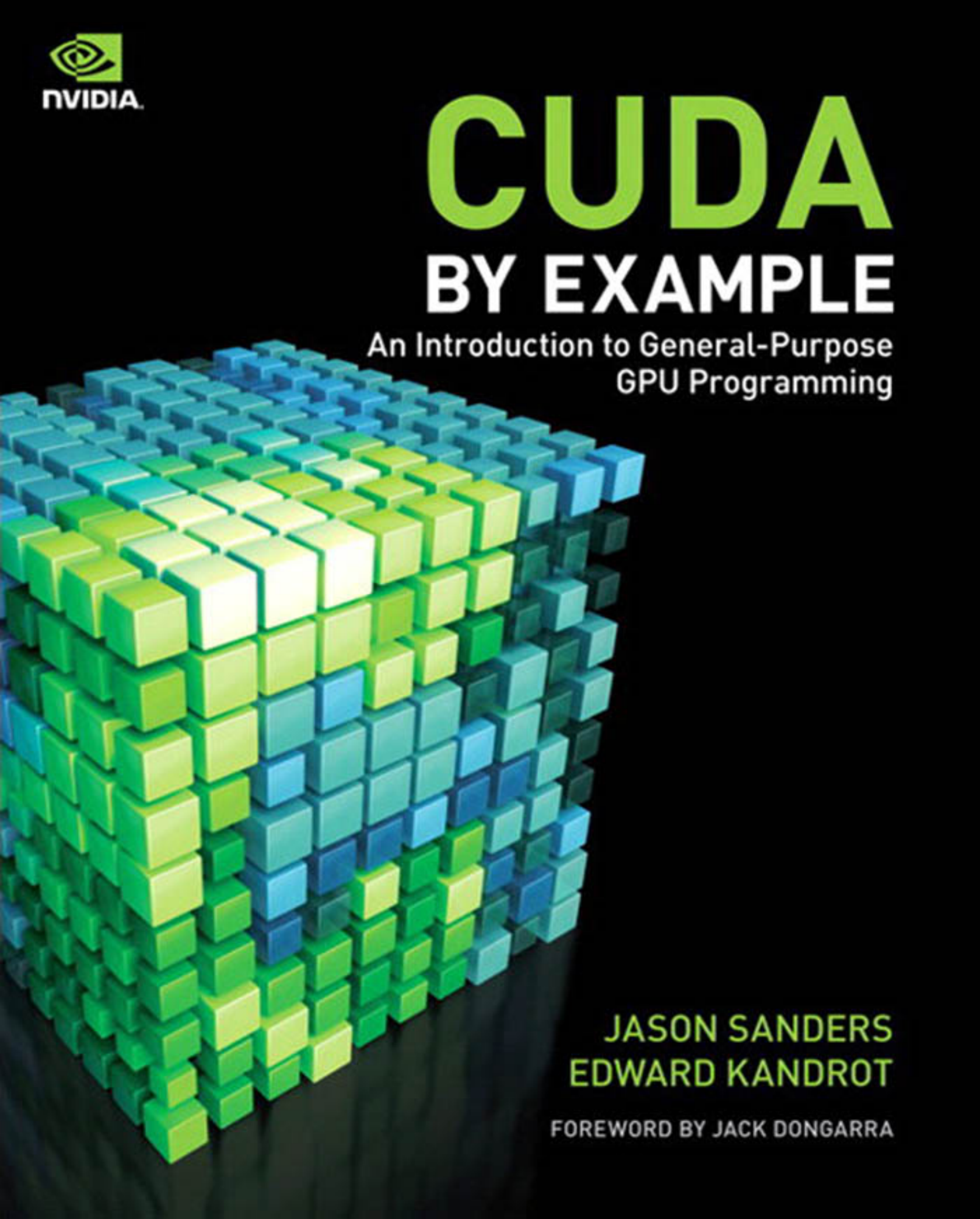
NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose
GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA

CUDA by Example

```
    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

function of `(x,y,t)` and store this value in the output buffer. In this case, the function produces a time-varying sinusoidal "ripple."

```
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));
```

We recommend that you not get too hung up on the computation of `grey`. It's essentially just a 2D function of time that makes a nice rippling effect when it's animated. A screenshot of one frame should look something like Figure 5.3.

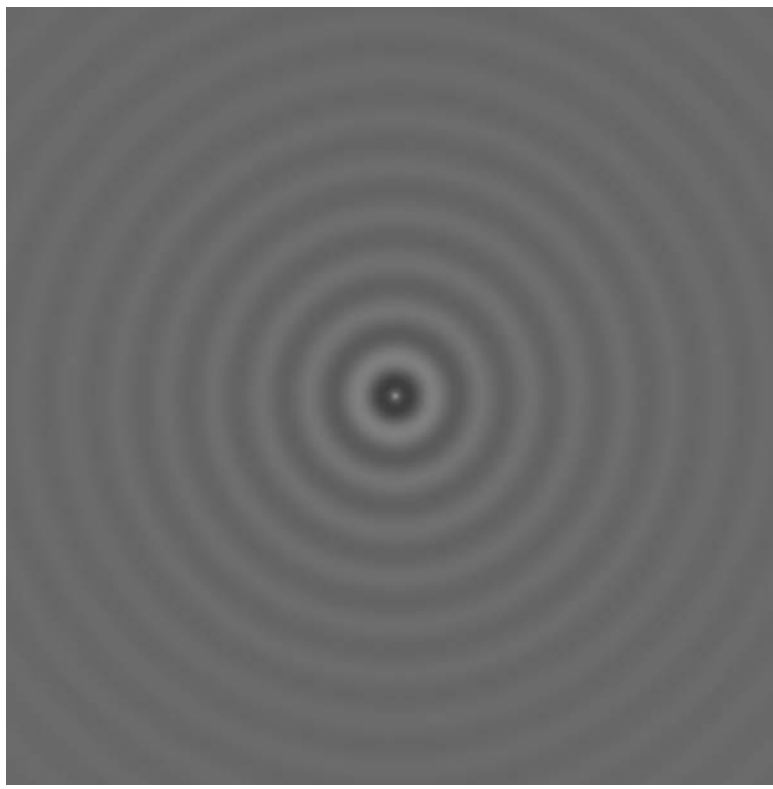

Figure 5.3 A screenshot from the GPU ripple example

5.3 Shared Memory and Synchronization

So far, the motivation for splitting blocks into threads was simply one of working around hardware limitations to the number of blocks we can have in flight. This is fairly weak motivation, because this could easily be done behind the scenes by the CUDA runtime. Fortunately, there are other reasons one might want to split a block into threads.

CUDA C makes available a region of memory that we call *shared memory*. This region of memory brings along with it another extension to the C language akin to `__device__` and `__global__`. As a programmer, you can modify your variable declarations with the CUDA C keyword `__shared__` to make this variable resident in shared memory. But what's the point?

We're glad you asked. The CUDA C compiler treats variables in shared memory differently than typical variables. It creates a copy of the variable for each block that you launch on the GPU. Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks. This provides an excellent means by which threads within a block can communicate and collaborate on computations. Furthermore, shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM. Because of this, the latency to access shared memory tends to be far lower than typical buffers, making shared memory effective as a per-block, software-managed cache or scratchpad.

The prospect of communication between threads should excite you. It excites us, too. But nothing in life is free, and interthread communication is no exception. If we expect to communicate between threads, we also need a mechanism for synchronizing between threads. For example, if thread A writes a value to shared memory and we want thread B to do something with this value, we can't have thread B start its work until we know the write from thread A is complete. Without synchronization, we have created a race condition where the correctness of the execution results depends on the nondeterministic details of the hardware.

Let's take a look at an example that uses these features.

5.3.1 DOT PRODUCT

Congratulations! We have graduated from vector addition and will now take a look at vector dot products (sometimes called an *inner product*). We will quickly review what a dot product is, just in case you are unfamiliar with vector mathematics (or it has been a few years). The computation consists of two steps. First, we multiply corresponding elements of the two input vectors. This is very similar to vector addition but utilizes multiplication instead of addition. However, instead of then storing these values to a third, output vector, we sum them all to produce a single scalar output.

For example, if we take the dot product of two four-element vectors, we would get Equation 5.1.

Equation 5.1

$$(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

Perhaps the algorithm we tend to use is becoming obvious. We can do the first step exactly how we did vector addition. Each thread multiplies a pair of corresponding entries, and then every thread moves on to its next pair. Because the result needs to be the sum of all these pairwise products, each thread keeps a running sum of the pairs it has added. Just like in the addition example, the threads increment their indices by the total number of threads to ensure we don't miss any elements and don't multiply a pair twice. Here is the first step of the dot product routine:

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

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
```

```
    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;
```

As you can see, we have declared a buffer of shared memory named `cache`. This buffer will be used to store each thread's running sum. Soon we will see *why* we do this, but for now we will simply examine the mechanics by which we accomplish it. It is trivial to declare a variable to reside in shared memory, and it is identical to the means by which you declare a variable as `static` or `volatile` in standard C:

```
__shared__ float cache[threadsPerBlock];
```

We declare the array of size `threadsPerBlock` so each thread in the block has a place to store its temporary result. Recall that when we have allocated memory globally, we allocated enough for every thread that runs the kernel, or `threadsPerBlock` times the total number of blocks. But since the compiler will create a copy of the shared variables for each block, we need to allocate only enough memory such that each thread in the block has an entry.

After allocating the shared memory, we compute our data indices much like we have in the past:

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

The computation for the variable `tid` should look familiar by now; we are just combining the block and thread indices to get a global offset into our input arrays. The offset into our shared memory cache is simply our thread index. Again, we don't need to incorporate our block index into this offset because each block has its own private copy of this shared memory.

Finally, we clear our shared memory buffer so that later we will be able to blindly sum the entire array without worrying whether a particular entry has valid data stored there:

```
// set the cache values
cache[cacheIndex] = temp;
```

It will be possible that not every entry will be used if the size of the input vectors is not a multiple of the number of threads per block. In this case, the last block will have some threads that do nothing and therefore do not write values.

Each thread computes a running sum of the product of corresponding entries in `a` and `b`. After reaching the end of the array, each thread stores its temporary sum into the shared buffer.

```
float   temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}

// set the cache values
cache[cacheIndex] = temp;
```

At this point in the algorithm, we need to sum all the temporary values we've placed in the cache. To do this, we will need some of the threads to read the values that have been stored there. However, as we mentioned, this is a potentially dangerous operation. We need a method to guarantee that all of these writes to the shared array `cache[]` complete before anyone tries to read from this buffer. Fortunately, such a method exists:

```
// synchronize threads in this block
__syncthreads();
```

This call guarantees that every thread in the block has completed instructions prior to the `__syncthreads()` before the hardware will execute the next