Chapter 6

Constant Memory and Events

We hope you have learned much about writing code that executes on the GPU. You should know how to spawn parallel blocks to execute your kernels, and you should know how to further split these blocks into parallel threads. You have also seen ways to enable communication and synchronization between these threads. But since the book is not over yet, you may have guessed that CUDA C has even more features that might be useful to you.

This chapter will introduce you to a couple of these more advanced features. Specifically, there exist ways in which you can exploit special regions of memory on your GPU in order to accelerate your applications. In this chapter, we will discuss one of these regions of memory: *constant memory*. In addition, because we are looking at our first method for enhancing the performance of your CUDA C applications, you will also learn how to measure the performance of your applications using CUDA *events*. From these measurements, you will be able to quantify the gain (or loss!) from any enhancements you make.

6.1 Chapter Objectives

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

- You will learn about using constant memory with CUDA C.
- You will learn about the performance characteristics of constant memory.
- You will learn how to use CUDA events to measure application performance.

6.2 Constant Memory

Previously, we discussed how modern GPUs are equipped with enormous amounts of arithmetic processing power. In fact, the computational advantage graphics processors have over CPUs helped precipitate the initial interest in using graphics processors for general-purpose computing. With hundreds of arithmetic units on the GPU, often the bottleneck is not the arithmetic throughput of the chip but rather the memory bandwidth of the chip. There are so many ALUs on graphics processors that sometimes we just can't keep the input coming to them fast enough to sustain such high rates of computation. So, it is worth investigating means by which we can reduce the amount of memory traffic required for a given problem.

We have seen CUDA C programs that have used both global and shared memory so far. However, the language makes available another kind of memory known as *constant memory*. As the name may indicate, we use constant memory for data that will not change over the course of a kernel execution. NVIDIA hardware provides 64KB of constant memory that it treats differently than it treats standard global memory. In some situations, using constant memory rather than global memory will reduce the required memory bandwidth.

6.2.1 RAY TRACING INTRODUCTION

We will look at one way of exploiting constant memory in the context of a simple *ray tracing* application. First, we will give you some background in the major concepts behind ray tracing. If you are already comfortable with the concepts behind ray tracing, you can skip to the "Ray Tracing on the GPU" section.

Simply put, ray tracing is one way of producing a two-dimensional image of a scene consisting of three-dimensional objects. But isn't this what GPUs were originally designed for? How is this different from what OpenGL or DirectX do when you play your favorite game? Well, GPUs do indeed solve this same problem, but they use a technique known as *rasterization*. There are many excellent books on rasterization, so we will not endeavor to explain the differences here. It suffices to say that they are completely different methods that solve the same problem.

So, how does ray tracing produce an image of a three-dimensional scene? The idea is simple: We choose a spot in our scene to place an imaginary camera. This simplified digital camera contains a light sensor, so to produce an image, we need to determine what light would hit that sensor. Each pixel of the resulting image should be the same color and intensity of the ray of light that hits that spot sensor.

Since light incident at any point on the sensor can come from any place in our scene, it turns out it's easier to work backward. That is, rather than trying to figure out what light ray hits the pixel in question, what if we imagine shooting a ray *from* the pixel and into the scene? In this way, each pixel behaves something like an eye that is "looking" into the scene. Figure 6.1 illustrates these rays being cast out of each pixel and into the scene.

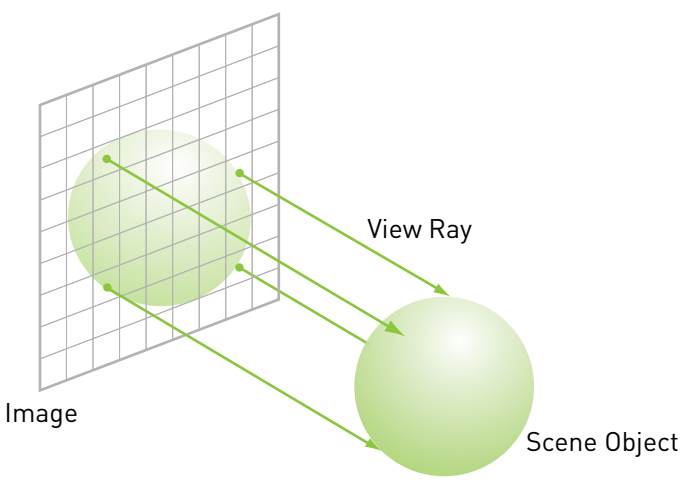

Figure 6.1 A simple ray tracing scheme

We figure out what color is seen by each pixel by tracing a ray from the pixel in question through the scene until it hits one of our objects. We then say that the pixel would "see" this object and can assign its color based on the color of the object it sees. Most of the computation required by ray tracing is in the computation of these intersections of the ray with the objects in the scene.

Moreover, in more complex ray tracing models, shiny objects in the scene can reflect rays, and translucent objects can refract the rays of light. This creates secondary rays, tertiary rays, and so on. In fact, this is one of the attractive features of ray tracing; it is very simple to get a basic ray tracer working, but we can build models of more complex phenomenon into the ray tracer in order to produce more realistic images.

6.2.2 RAY TRACING ON THE GPU

Since APIs such as OpenGL and DirectX are not designed to allow ray-traced rendering, we will have to use CUDA C to implement our basic ray tracer. Our ray tracer will be extraordinarily simple so that we can concentrate on the use of constant memory, so if you were expecting code that could form the basis of a full-blown production renderer, you will be disappointed. Our basic ray tracer will only support scenes of spheres, and the camera is restricted to the z-axis, facing the origin. Moreover, we will not support any lighting of the scene to avoid the complications of secondary rays. Instead of computing lighting effects, we will simply assign each sphere a color and then shade them with some precomputed function if they are visible.

So, what *will* the ray tracer do? It will fire a ray from each pixel and keep track of which rays hit which spheres. It will also track the depth of each of these hits. In the case where a ray passes through multiple spheres, only the sphere closest to the camera can be seen. In essence, our "ray tracer" is not doing much more than hiding surfaces that cannot be seen by the camera.

We will model our spheres with a data structure that stores the sphere's center coordinate of `(x, y, z)`, its `radius`, and its color of `(r, b, g)`.

```
#define INF     2e10f

struct Sphere {
    float   r,b,g;
    float   radius;
    float   x,y,z;
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
```

You will also notice that the structure has a method called `hit( float ox, float oy, float *n )`. Given a ray shot from the pixel at `(ox, oy)`, this method computes whether the ray intersects the sphere. If the ray *does* intersect the sphere, the method computes the distance from the camera where the ray hits the sphere. We need this information for the reason mentioned before: In the event that the ray hits more than one sphere, only the closest sphere can actually be seen.

Our `main()` routine follows roughly the same sequence as our previous image-generating examples.

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

#define rnd( x ) (x * rand() / RAND_MAX)
#define SPHERES 20

Sphere  *s;
```

```
int main( void ) {
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );
    // allocate memory for the Sphere dataset
    HANDLE_ERROR( cudaMalloc( (void**)&s,
                              sizeof(Sphere) * SPHERES ) );
```

We allocate memory for our input data, which is an array of spheres that compose our scene. Since we need this data on the GPU but are generating it with the CPU, we have to do both a `cudaMalloc()` *and* a `malloc()` to allocate memory on both the GPU and the CPU. We also allocate a bitmap image that we will fill with output pixel data as we ray trace our spheres on the GPU.

After allocating memory for input and output, we randomly generate the center coordinate, color, and radius for our spheres:

```
    // allocate temp memory, initialize it, copy to
    // memory on the GPU, and then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
```

The program currently generates a random array of 20 spheres, but this quantity is specified in a `#define` and can be adjusted accordingly.

We copy this array of spheres to the GPU using `cudaMemcpy()` and then free the temporary buffer.

```
    HANDLE_ERROR( cudaMemcpy( s, temp_s,
                              sizeof(Sphere) * SPHERES,
                              cudaMemcpyHostToDevice ) );
    free( temp_s );
```

Now that our input is on the GPU and we have allocated space for the output, we are ready to launch our kernel.

```
    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );
```

We will examine the kernel itself in a moment, but for now you should take it on faith that it ray traces the scene and generates pixel data for the input scene of spheres. Finally, we copy the output image back from the GPU and display it. It should go without saying that we free all allocated memory that hasn't already been freed.

```
    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    // free our memory
    cudaFree( dev_bitmap );
    cudaFree( s );
}
```

All of this should be commonplace to you now. So, how do we do the actual ray tracing? Because we have settled on a very simple ray tracing model, our kernel will be very easy to understand. Each thread is generating one pixel for our output image, so we start in the usual manner by computing the `x`- and `y`-coordinates for the thread as well as the linearized `offset` into our output buffer. We will also shift our `(x,y)` image coordinates by `DIM/2` so that the z-axis runs through the center of the image.

```
__global__ void kernel( unsigned char *ptr ) {
    // 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;
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);
```

Since each ray needs to check each sphere for intersection, we will now iterate through the array of spheres, checking each for a hit.

```
    float   r=0, g=0, b=0;
    float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n;
        float   t = s[i].hit( ox, oy, &n );
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
        }
    }
```

Clearly, the majority of the interesting computation lies in the `for()` loop. We iterate through each of the input spheres and call its `hit()` method to determine whether the ray from our pixel "sees" the sphere. If the ray hits the current sphere, we determine whether the hit is closer to the camera than the last sphere we hit. If it is closer, we store this depth as our new closest sphere. In addition, we

store the color associated with this sphere so that when the loop has terminated, the thread knows the color of the sphere that is closest to the camera. Since this is the color that the ray from our pixel "sees," we conclude that this is the color of the pixel and store this value in our output image buffer.

After every sphere has been checked for intersection, we can store the current color into the output image.

```
        ptr[offset*4 + 0] = (int)(r * 255);
        ptr[offset*4 + 1] = (int)(g * 255);
        ptr[offset*4 + 2] = (int)(b * 255);
        ptr[offset*4 + 3] = 255;
    }
```

Note that if no spheres have been hit, the color that we store will be whatever color we initialized the variables `r`, `b`, and `g` to. In this case, we set `r`, `b`, and `g` to zero so the background will be black. You can change these values to render a different color background. Figure 6.2 shows an example of what the output should look like when rendered with 20 spheres and a black background.

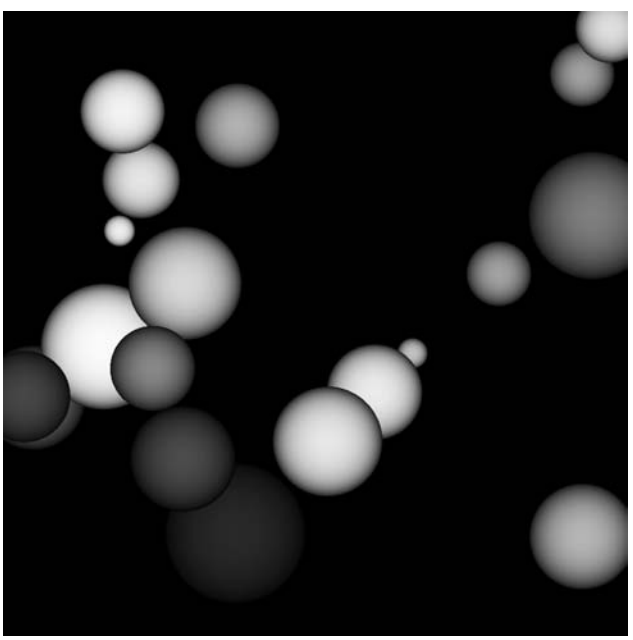

Figure 6.2 A screenshot from the ray tracing example

Since we randomly generated the sphere positions, colors, and sizes, we advise you not to panic if your output doesn't match this image identically.

6.2.3 RAY TRACING WITH CONSTANT MEMORY

You may have noticed that we never mentioned constant memory in the ray tracing example. Now it's time to improve this example using the benefits of constant memory. Since we cannot modify constant memory, we clearly can't use it for the output image data. And this example has only one input, the array of spheres, so it should be pretty obvious what data we will store in constant memory.

The mechanism for declaring memory constant is identical to the one we used for declaring a buffer as shared memory. Instead of declaring our array like this:

```
Sphere  *s;
```

we add the modifier `__constant__` before it:

```
__constant__ Sphere s[SPHERES];
```

Notice that in the original example, we declared a pointer and then used `cudaMalloc()` to allocate GPU memory for it. When we changed it to constant memory, we also changed the declaration to statically allocate the space in constant memory. We no longer need to worry about calling `cudaMalloc()` or `cudaFree()` for our array of spheres, but we do need to commit to a size for this array at compile-time. For many applications, this is an acceptable trade-off for the performance benefits of constant memory. We will talk about these benefits momentarily, but first we will look at how the use of constant memory changes our `main()` routine:

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

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );
```

```
    // allocate temp memory, initialize it, copy to constant
    // memory on the GPU, and then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                      sizeof(Sphere) * SPHERES) );
    free( temp_s );

    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    // free our memory
    cudaFree( dev_bitmap );
}
```

Largely, this is identical to the previous implementation of `main()`. As we mentioned previously, we no longer need the call to `cudaMalloc()` to allocate

space for our array of spheres. The other change has been highlighted in the listing:

```
    HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                      sizeof(Sphere) * SPHERES ) );
```

We use this special version of `cudaMemcpy()` when we copy from host memory to constant memory on the GPU. The only differences between `cudaMemcpyToSymbol()` and `cudaMemcpy()` using `cudaMemcpyHostToDevice` are that `cudaMemcpyToSymbol()` copies to constant memory and `cudaMemcpy()` copies to global memory.

Outside the `__constant__` modifier and the two changes to `main()`, the versions with and without constant memory are identical.

6.2.4 PERFORMANCE WITH CONSTANT MEMORY

Declaring memory as `__constant__` constrains our usage to be read-only. In taking on this constraint, we expect to get something in return. As we previously mentioned, reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory. There are two reasons why reading from the 64KB of constant memory can save bandwidth over standard reads of global memory:

- A single read from constant memory can be broadcast to other "nearby" threads, effectively saving up to 15 reads.
- Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic.

What do we mean by the word *nearby*? To answer this question, we will need to explain the concept of a *warp*. For those readers who are more familiar with *Star Trek* than with weaving, a warp in this context has nothing to do with the speed of travel through space. In the world of weaving, a warp refers to the group of *threads* being woven together into fabric. In the CUDA Architecture, a *warp* refers to a collection of 32 threads that are "woven together" and get executed in lockstep. At every line in your program, each thread in a warp executes the same instruction on different data.

When it comes to handling constant memory, NVIDIA hardware can broadcast a single memory read to each half-warp. A half-warp—not nearly as creatively named as a warp—is a group of 16 threads: half of a 32-thread warp. If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread. If you are reading a lot of data from constant memory, you will generate only 1/16 (roughly 6 percent) of the memory traffic as you would when using global memory.

But the savings don't stop at a 94 percent reduction in bandwidth when reading constant memory! Because we have committed to leaving the memory unchanged, the hardware can aggressively cache the constant data on the GPU. So after the first read from an address in constant memory, other half-warps requesting the same address, and therefore hitting the constant cache, will generate no additional memory traffic.

In the case of our ray tracer, every thread in the launch reads the data corresponding to the first sphere so the thread can test its ray for intersection. After we modify our application to store the spheres in constant memory, the hardware needs to make only a single request for this data. After caching the data, every other thread avoids generating memory traffic as a result of one of the two constant memory benefits:

- It receives the data in a half-warp broadcast.
- It retrieves the data from the constant memory cache.

Unfortunately, there can potentially be a downside to performance when using constant memory. The half-warp broadcast feature is in actuality a double-edged sword. Although it can dramatically accelerate performance when all 16 threads are reading the same address, it actually slows performance to a crawl when all 16 threads read different addresses.

The trade-off to allowing the broadcast of a single read to 16 threads is that the 16 threads are allowed to place only a single read request at a time. For example, if all 16 threads in a half-warp need different data from constant memory, the 16 different reads get serialized, effectively taking 16 times the amount of time to place the request. If they were reading from conventional global memory, the request could be issued at the same time. In this case, reading from constant memory would probably be slower than using global memory.

6.3 Measuring Performance with Events

Fully aware that there may be either positive or negative implications, you have changed your ray tracer to use constant memory. How do you determine how this has impacted the performance of your program? One of the simplest metrics involves answering this simple question: Which version takes less time to finish? We could use one of the CPU or operating system timers, but this will include latency and variation from any number of sources (operating system thread scheduling, availability of high-precision CPU timers, and so on). Furthermore, while the GPU kernel runs, we may be asynchronously performing computation on the host. The only way to time these host computations is using the CPU or operating system timing mechanism. So to measure the time a GPU spends on a task, we will use the CUDA event API.

An *event* in CUDA is essentially a GPU time stamp that is recorded at a user-specified point in time. Since the GPU itself is recording the time stamp, it eliminates a lot of the problems we might encounter when trying to time GPU execution with CPU timers. The API is relatively easy to use, since taking a time stamp consists of just two steps: creating an event and subsequently recording an event. For example, at the beginning of some sequence of code, we instruct the CUDA runtime to make a record of the current time. We do so by creating and then recording the event:

```
cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord( start, 0 );
```

You will notice that when we instruct the runtime to record the event `start`, we also pass it a second argument. In the previous example, this argument is 0. The exact nature of this argument is unimportant for our purposes right now, so we intend to leave it mysteriously unexplained rather than open a new can of worms. If your curiosity is killing you, we intend to discuss this when we talk about *streams*.

To time a block of code, we will want to create both a start event and a stop event. We will have the CUDA runtime record when we start tell it to do some other work on the GPU and then tell it to record when we've stopped:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

// do some work on the GPU

cudaEventRecord( stop, 0 );
```

Unfortunately, there is still a problem with timing GPU code in this way. The fix will require only one line of code but will require some explanation. The trickiest part of using events arises as a consequence of the fact that some of the calls we make in CUDA C are actually *asynchronous*. For example, when we launched the kernel in our ray tracer, the GPU begins executing our code, but the CPU continues executing the next line of our program before the GPU finishes. This is excellent from a performance standpoint because it means we can be computing something on the GPU and CPU at the same time, but conceptually it makes timing tricky.

You should imagine calls to `cudaEventRecord()` as an instruction to record the current time being placed into the GPU's pending queue of work. As a result, our event won't actually be recorded until the GPU finishes everything prior to the call to `cudaEventRecord()`. In terms of having our `stop` event measure the correct time, this is precisely what we want. But we cannot safely *read* the value of the `stop` event until the GPU has completed its prior work and recorded the `stop` event. Fortunately, we have a way to instruct the CPU to synchronize on an event, the event API function `cudaEventSynchronize()`:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

// do some work on the GPU

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
```

Now, we have instructed the runtime to block further instruction until the GPU has reached the `stop` event. When the call to `cudaEventSynchronize()`

returns, we know that all GPU work before the `stop` event has completed, so it is safe to read the time stamp recorded in `stop`. It is worth noting that because CUDA events get implemented directly on the GPU, they are unsuitable for timing mixtures of device and host code. That is, you will get unreliable results if you attempt to use CUDA events to time more than kernel executions and memory copies involving the device.

6.3.1 MEASURING RAY TRACER PERFORMANCE

To time our ray tracer, we will need to create a start and stop event, just as we did when learning about events. The following is a timing-enabled version of the ray tracer that does *not* use constant memory:

```
int main( void ) {
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );
    // allocate memory for the Sphere dataset
    HANDLE_ERROR( cudaMalloc( (void**)&s,
                              sizeof(Sphere) * SPHERES ) );

    // allocate temp memory, initialize it, copy to
    // memory on the GPU, and then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
```

```
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpy( s, temp_s,
                              sizeof(Sphere) * SPHERES,
                              cudaMemcpyHostToDevice ) );
    free( temp_s );

    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( s, dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );

    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    // display
    bitmap.display_and_exit();

    // free our memory
    cudaFree( dev_bitmap );
    cudaFree( s );
}
```

Notice that we have thrown two additional functions into the mix, the calls to `cudaEventElapsedTime()` and `cudaEventDestroy()`. The function `cudaEventElapsedTime()` is a utility that computes the elapsed time between two previously recorded events. The time in milliseconds elapsed between the two events is returned in the first argument, the address of a floating-point variable.

The call to `cudaEventDestroy()` needs to be made when we're finished using an event created with `cudaEventCreate()`. This is identical to calling `free()` on memory previously allocated with `malloc()`, so we needn't stress how important it is to match every `cudaEventCreate()` with a `cudaEventDestroy()`.

We can instrument the ray tracer that does use constant memory in the same fashion:

```
int main( void ) {
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );

    // allocate temp memory, initialize it, copy to constant
    // memory on the GPU, and then free our temp memory
   Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
```

```
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                      sizeof(Sphere) * SPHERES) );
    free( temp_s );

    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    // display
    bitmap.display_and_exit();

    // free our memory
    cudaFree( dev_bitmap );
}
```

Now when we run our two versions of the ray tracer, we can compare the time it takes to complete the GPU work. This will tell us at a high level whether introducing constant memory has improved the performance of our application or worsened it. Fortunately, in this case, performance is improved dramatically by using constant memory. Our experiments on a GeForce GTX 280 show the constant memory ray tracer performing up to 50 percent faster than the version that uses global memory. On a different GPU, your mileage might vary, although the ray tracer that uses constant memory should always be at least as fast as the version without it.

6.4 Chapter Review

In addition to the global and shared memory we explored in previous chapters, NVIDIA hardware makes other types of memory available for our use. Constant memory comes with additional constraints over standard global memory, but in some cases, subjecting ourselves to these constraints can yield additional performance. Specifically, we can see additional performance when threads in a warp need access to the same read-only data. Using constant memory for data with this access pattern can conserve bandwidth both because of the capacity to broadcast reads across a half-warp and because of the presence of a constant memory cache on chip. Memory bandwidth bottlenecks a wide class of algorithms, so having mechanisms to ameliorate this situation can prove incredibly useful.

We also learned how to use CUDA events to request the runtime to record time stamps at specific points during GPU execution. We saw how to synchronize the CPU with the GPU on one of these events and then how to compute the time elapsed between two events. In doing so, we built up a method to compare the running time between two different methods for ray tracing spheres, concluding that, for the application at hand, using constant memory gained us a significant amount of performance.