Ray Tracing in One Weekend

Contained documents identified by InYoung Cho (조인영):

Ray Tracing in One Weekend

http://in1weekend.blogspot.com/2016/01/ray-tracing-in-one-weekend.html

Accelerated Ray Tracing in One Weekend in CUDA

https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/

An Even Easier Introduction to CUDA

https://devblogs.nvidia.com/even-easier-introduction-cuda/

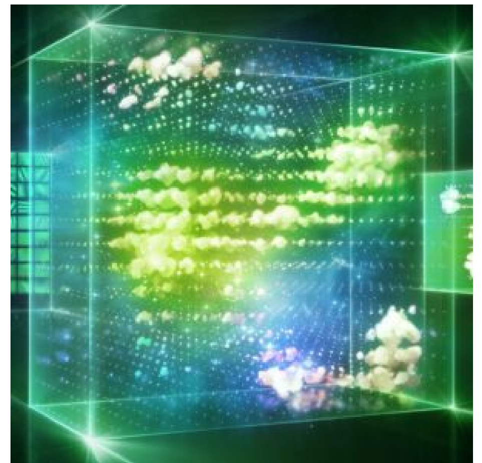
Commnets from InYoung Cho (조인영)

첫번째 자료는 두번째 자료를 읽다가 필요할 때마다 링크를 타고 들어가서 필요한 코드만 따로 보는게 더 좋을 것 같았습니다.

ACCELERATED COMPUTING ARTIFICIAL INTELLIGENCE

An Even Easier Introduction to CUDA

By Mark Harris | January 25, 2017 Tags: Accelerated Computing, Beginner, CUDA, Development Tools and Libraries, machine learning and AI



This post is a super simple introduction to CUDA, the popular parallel computing platform and programming model from NVIDIA. I wrote a previous "Easy Introduction" to CUDA in 2013 that has been very popular over the years. But CUDA programming has gotten easier, and GPUs have gotten much faster, so it's time for an updated (and even easier) introduction.

CUDA C++ is just one of the ways you can create massively parallel applications with CUDA. It lets you use the powerful C++ programming language to develop high performance algorithms accelerated by thousands of parallel threads running on GPUs. Many developers have accelerated their computation- and bandwidth-hungry applications this way, including the libraries and frameworks that underpin the ongoing revolution in artificial intelligence known as Deep Learning.

So, you've heard about CUDA and you are interested in learning how to use it in your own applications. If you are a C or C++ programmer, this blog post should give you a good start. To follow along, you'll need a computer with an CUDA-capable GPU (Windows, Mac, or Linux, and any NVIDIA GPU should do), or a cloud instance with GPUs (AWS, Azure, IBM SoftLayer, and other cloud service providers have them). You'll also need the free CUDA Toolkit installed.

Let's get started!

Starting Simple

We'll start with a simple C++ program that adds the elements of two arrays with a million elements each.

1.4k
Shares

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
```

First, compile and run this C++ program. Put the code above in a file and save it as `add.cpp`, and then compile it with your C++ compiler. I'm on a Mac so I'm using `clang++`, but you can use `g++` on Linux or MSVC on Windows.

```
> clang++ add.cpp -o add
```

Then run it:

```
> ./add
 Max error: 0.000000
```

(On Windows you may want to name the executable add.exe and run it with `.\add`.)

As expected, it prints that there was no error in the summation and then exits. Now I want to get this computation running (in parallel) on the many cores of a GPU. It's actually pretty easy to take the first steps.

First, I just have to turn our `add` function into a function that the GPU can run, called a *kernel* in CUDA. To do this, all I have to do is add the specifier `__global__` to the function, which tells the CUDA C++ compiler that this is a function that runs on the GPU and can be called from CPU code.

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

These `__global__` functions are known as *kernels*, and code that runs on the GPU is often called *device code*, while code that runs on the CPU is *host code*.

Memory Allocation in CUDA

To compute on the GPU, I need to allocate memory accessible by the GPU. Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system. To allocate data in unified memory, call `cudaMallocManaged()`, which returns a pointer that you can access from host (CPU) code or device (GPU) code. To free the data, just pass the pointer to `cudaFree()`.

I just need to replace the calls to `new` in the code above with calls to `cudaMallocManaged()`, and replace calls to `delete []` with calls to `cudaFree.`

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

Finally, I need to *launch* the `add()` kernel, which invokes it on the GPU. CUDA kernel launches are specified using the triple angle bracket syntax <<< >>>. I just have to add it to the call to `add` before the parameter list.

```
add<<<1, 1>>>(N, x, y);
```

Easy! I'll get into the details of what goes inside the angle brackets soon; for now all you need to know is that this line launches one GPU thread to run `add()`.

Just one more thing: I need the CPU to wait until the kernel is done before it accesses the results (because CUDA kernel launches don't block the calling CPU thread). To do this I just call `cudaDeviceSynchronize()` before doing the final error checking on the CPU.

Here's the complete code:

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
```

CUDA files have the file extension `.cu`. So save this code in a file called `add.cu` and compile it with `nvcc`, the CUDA C++ compiler.

```
> nvcc add.cu -o add_cuda
> ./add_cuda
Max error: 0.000000
```

This is only a first step, because as written, this kernel is only correct for a single thread, since every thread that runs it will perform the add on the whole array. Moreover, there is a race condition since multiple parallel threads would both read and write the same locations.

Note: on Windows, you need to make sure you set Platform to x64 in the Configuration Properties for your project in Microsoft Visual Studio.

Profile it!

I think the simplest way to find out how long the kernel takes to run is to run it with `nvprof`, the command line GPU profiler

that comes with the CUDA Toolkit. Just type `nvprof ./add_cuda` on the command line:

```
$ nvprof ./add_cuda
==3355== NVPROF is profiling process 3355, command: ./add_cuda
Max error: 0
==3355== Profiling application: ./add_cuda
==3355== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  463.25ms         1  463.25ms  463.25ms  463.25ms  add(int, float*, float*)
...
```

Above is the truncated output from `nvprof`, showing a single call to `add`. It takes about half a second on an NVIDIA Tesla K80 accelerator, and about the same time on an NVIDIA GeForce GT 740M in my 3-year-old Macbook Pro.

Let's make it faster with parallelism.

Picking up the Threads

Now that you've run a kernel with one thread that does some computation, how do you make it parallel? The key is in CUDA's `<<<1, 1>>>` syntax. This is called the execution configuration, and it tells the CUDA runtime how many parallel threads to use for the launch on the GPU. There are two parameters here, but let's start by changing the second one: the number of threads in a thread block. CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose.

```
add<<<1, 256>>>(N, x, y);
```

If I run the code with only this change, it will do the computation once per thread, rather than spreading the computation across the parallel threads. To do it properly, I need to modify the kernel. CUDA C++ provides keywords that let kernels get the indices of the running threads. Specifically, `threadIdx.x` contains the index of the current thread within its block, and `blockDim.x` contains the number of threads in the block. I'll just modify the loop to stride through the array with parallel threads.

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

The `add` function hasn't changed that much. In fact, setting `index` to 0 and `stride` to 1 makes it semantically identical to the first version.

Save the file as `add_block.cu` and compile and run it in `nvprof` again. For the remainder of the post I'll just show the relevant line from the output.

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  2.7107ms         1  2.7107ms  2.7107ms  2.7107ms  add(int, float*, float*)
```

That's a big speedup (463ms down to 2.7ms), but not surprising since I went from 1 thread to 256 threads. The K80 is faster than my little Macbook Pro GPU (at 3.2ms). Let's keep going to get even more performance.

Out of the Blocks

CUDA GPUs have many parallel processors grouped into Streaming Multiprocessors, or SMs. Each SM can run multiple concurrent thread blocks. As an example, a Tesla P100 GPU based on the Pascal GPU Architecture has 56 SMs, each capable of supporting up to 2048 active threads. To take full advantage of all these threads, I should launch the kernel with multiple thread blocks.

By now you may have guessed that the first parameter of the execution configuration specifies the number of thread blocks. Together, the blocks of parallel threads make up what is known as the *grid*. Since I have `N` elements to process, and 256 threads per block, I just need to calculate the number of blocks to get at least N threads. I simply divide `N` by the block size (being careful to round up in case `N` is not a multiple of `blockSize`).

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

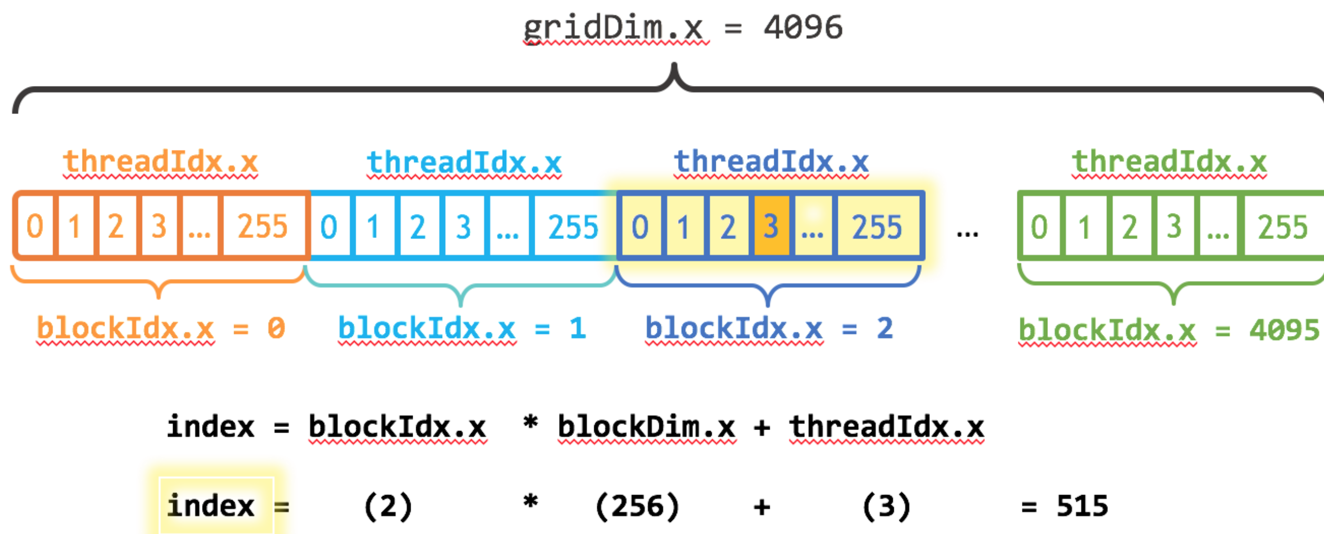


Figure 1: The CUDA parallel thread hierarchy. CUDA executes kernels using a *grid* of *blocks* of *threads*. This figure shows the common indexing pattern used in CUDA programs using the CUDA keywords gridDim.x (the number of thread blocks), blockDim.x (the number of threads in each block), blockIdx.x (the index the current block within the grid), and threadIdx.x (the index of the current thread within the block).

I also need to update the kernel code to take into account the entire grid of thread blocks. CUDA provides `gridDim.x`, which contains the number of blocks in the grid, and `blockIdx.x`, which contains the index of the current thread block in the grid. Figure 1 illustrates the the approach to indexing into an array (one-dimensional) in CUDA using `blockDim.x`, `gridDim.x`, and `threadIdx.x`. The idea is that each thread gets its index by computing the offset to the beginning of its block (the block index times the block size: `blockIdx.x * blockDim.x`) and adding the thread's index within the block (`threadIdx.x`). The code `blockIdx.x * blockDim.x + threadIdx.x` is idiomatic CUDA.

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

The updated kernel also sets `stride` to the total number of threads in the grid (`blockDim.x * gridDim.x`). This type of loop in a CUDA kernel is often called a *grid-stride loop*.

Save the file as `add_grid.cu` and compile and run it in `nvprof` again.

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  94.015us         1  94.015us  94.015us  94.015us  add(int, float*, float*)
```

That's another 28x speedup, from running multiple blocks on all the SMs of a K80! We're only using one of the 2 GPUs on the K80, but each GPU has 13 SMs. Note the GeForce in my laptop has 2 (weaker) SMs and it takes 680us to run the kernel.

Summing Up

Here's a rundown of the performance of the three versions of the `add()` kernel on the Tesla K80 and the GeForce GT 750M.

	Laptop (GeForce GT 750M)		Server (Tesla K80)	
Version	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s

As you can see, we can achieve very high bandwidth on GPUs. The computation in this post is very bandwidth-bound, but GPUs also excel at heavily compute-bound computations such as dense matrix linear algebra, deep learning, image and signal processing, physical simulations, and more.

Excercises

To keep you going, here are a few things to try on your own. Please post about your experience in the comments section below.

1. Browse the CUDA Toolkit documentation. If you haven't installed CUDA yet, check out the Quick Start Guide and the installation guides. Then browse the Programming Guideand the Best Practices Guide. There are also tuning guides for various architectures.
2. Experiment with `printf()` inside the kernel. Try printing out the values of `threadIdx.x` and `blockIdx.x` for some or all of the threads. Do they print in sequential order? Why or why not?
3. Print the value of `threadIdx.y` or `threadIdx.z` (or `blockIdx.y`) in the kernel. (Likewise for `blockDim` and `gridDim`). Why do these exist? How do you get them to take on values other than 0 (1 for the dims)?
4. If you have access to a Pascal-based GPU, try running `add_grid.cu` on it. Is performance better or worse than the K80 results? Why? (Hint: read about Pascal's Page Migration Engine and the CUDA 8 Unified Memory API.) For a detailed answer to this question, see the post Unified Memory for CUDA Beginners.

Where To From Here?

I hope that this post has whet your appetite for CUDA and that you are interested in learning more and applying CUDA C++ in your own computations. If you have questions or comments, don't hesitate to reach out using the comments section below.

I plan to follow up this post with further CUDA programming material, but to keep you busy for now, there is a whole series of older introductory posts that you can continue with (and that I plan on updating / replacing in the future as needed):

- How to Implement Performance Metrics in CUDA C++
- How to Query Device Properties and Handle Errors in CUDA C++
- How to Optimize Data Transfers in CUDA C++
- How to Overlap Data Transfers in CUDA C++
- How to Access Global Memory Efficiently in CUDA C++
- Using Shared Memory in CUDA C++
- An Efficient Matrix Transpose in CUDA C++
- Finite Difference Methods in CUDA C++, Part 1
- Finite Difference Methods in CUDA C++, Part 2
- Accelerated Ray Tracing in One Weekend with CUDA

There is also a series of CUDA Fortran posts mirroring the above, starting with An Easy Introduction to CUDA Fortran.

You might also be interested in signing up for the online course on CUDA programming from Udacity and NVIDIA.

There is a wealth of other content on CUDA C++ and other GPU computing topics here on the NVIDIA Parallel Forall developer blog, so look around!

103 Comments

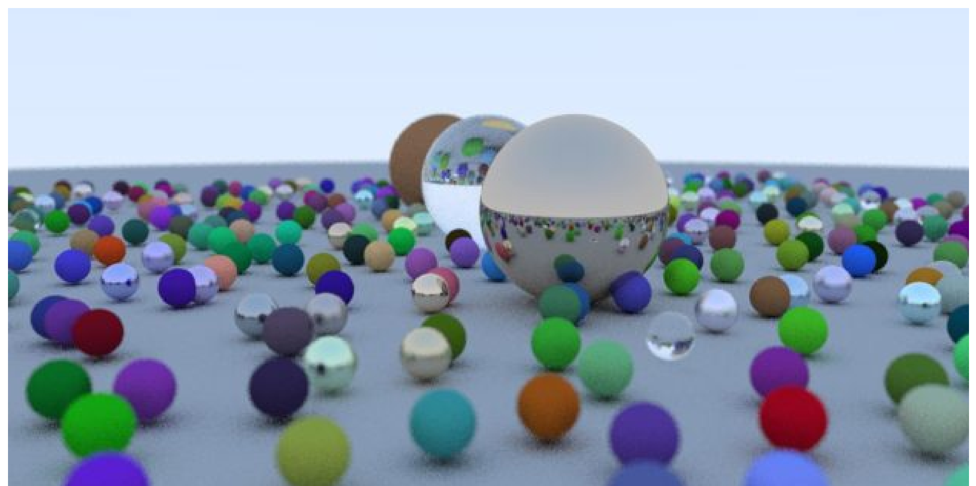
About the Authors

About Mark Harris
Mark is a Principal System Software Engineer working on RAPIDS. Mark has twenty years of experience developing software for GPUs, ranging from graphics and games, to physically-based simulation, to parallel algorithms and high-performance computing. While a Ph.D. student at The University of North Carolina he recognized a nascent trend and coined a name for it:

ACCELERATED COMPUTING DESIGN & VISUALIZATION

Accelerated Ray Tracing in One Weekend in CUDA

By Roger Allen | November 5, 2018 Tags: Accelerated Computing, C++, CUDA, Game Development, ray tracing



Recent announcements of NVIDIA's new Turing GPUs, RTX technology, and Microsoft's DirectX Ray Tracing have spurred a renewed interest in ray tracing. Using these technologies vastly simplifies the ability to write applications using ray tracing.

But what if you're curious about how ray tracing actually works? One way to learn is to code your own ray tracing engine. Would you like to build a ray tracer that runs on your GPU using CUDA? If so, this post is for you! You'll learn more about CUDA programming as well as ray tracing in one fell swoop.

Peter Shirley has written a series of fantastic ebooks about Ray Tracing starting from coding the very basics in one weekend to deep topics to spend your life investigating. You can find out more about these books at http://in1weekend.blogspot.com/. The books are now free or pay-what-you-wish and 50% of the proceeds go towards not-for-profit programming education organizations. They are also available on Amazon as a Kindle download. You should sit down and read *Ray Tracing in One Weekend* before diving into the rest of this post. Each section of this post corresponds to one of the chapters from the book. Even if you don't sit down and write your own ray tracer in C++, the core concepts should get you started with a GPU-based engine using CUDA.

Preliminaries

The C++ ray tracing engine in the One Weekend book is by no means the fastest ray tracer, but translating your C++ code to CUDA can result in a 10x or more speed improvement! Let's walk through the process of converting the C++ code from *Ray Tracing in One Weekend* to CUDA. Note that as you go through the C++ coding process, consider using git tags or branches to allow you to go back to each chapter's code easily. You can compare your C++ code with Peter Shirley's at https://github.com/petershirley/raytracinginoneweekend. After trying your hand at using CUDA, you can also compare with my CUDA code at https://github.com/rogerallen/raytracinginoneweekendincuda. Be sure to use the branches I've created for each chapter. (E.g. `git checkout ch12_where_next_cuda`.)

This post assumes you understand a few of the basics of CUDA. If not, you can start with An Even Easier Introduction to CUDA here on the NVIDIA Developer Blog. You'll need to have your development environment set up to compile and run CUDA code. We will also be profiling with `nvprof`, so you may want to familiarize yourself with how to profile your code with `nvprof`, too. The code in my repository was written using Ubuntu Linux and CUDA 9.x, but you should be able to adapt these instructions to recent CUDA releases on either Windows or MacOS, too.

1.4k
Shares

I build for a GTX 1070 card using specific `-gencode` flags for that card (`-gencode`

arch=compute_60,code=sm_60). You will want to adjust the architecture and feature settings for the GPU or GPUs you will be running on. In my Makefile, the main targets you will use are for building the executable make cudart and for running and creating the output image make out.ppm .

Please note that when the CUDA code runs for longer than a few seconds, you may notice error 6 (cudaErrorLaunchTimeout) and on Windows your screen may also black out for a few seconds. This is because your window manager thinks the GPU is malfunctioning when it doesn't respond after a certain amount of time. Linux users can take steps to resolve this issue via this post.

In this post, you'll learn about the following:

- a Unified Memory frame buffer that is written by the GPU and read by the CPU
- launching rendering work on the GPU
- Writing C++ code that can run on either the CPU or GPU
- Checking for slower double precision floating point code.
- C++ Memory management. Allocating memory on the GPU and instantiating it at run time on the GPU.
- Per-thread random number generation with cuRAND.

First Image

Chapter 1 in *Ray Tracing in One Weekend* ends with generating an image with a simple gradient for red & green channels. In a serial language, you use nested for loops to iterate over all of the pixels. In CUDA, the scheduler takes blocks of threads and schedules them on the GPU. But, before we get to that, we have to set up a few preliminaries…

For clarity, we change main.cc to main.cu to let both us and the compiler know this is CUDA source.

Each CUDA API call we make returns an error code that we should check. We check the cudaError_t result with a checkCudaErrors macro to output the error to stdout before we reset the CUDA device and exit. For help with any errors, see the cudaError_t enum section of https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__TYPES.html for what each error code means.

```
#define checkCudaErrors(val) check_cuda( (val), #val, __FILE__, __LINE__ )
void check_cuda(cudaError_t result, char const *const func, const char *const file, int const line) {
    if (result) {
        std::cerr << "CUDA error = " << static_cast<unsigned int>(result) << " at " <<
        file << ":" << line << " '" << func << "' \n";
        // Make sure we call CUDA Device Reset before exiting
        cudaDeviceReset();
        exit(99);
    }
}
```

We allocate an nx*ny image-sized frame buffer (FB) on the host to hold the RGB float values calculated by the GPU, allowing communication between the CPU and GPU. cudaMallocManaged allocates Unified Memory, which allows us to rely on the CUDA runtime to move the frame buffer on demand to the GPU for the rendering and back to the CPU for outputting the PPM image. Using cudaDeviceSynchronize lets the CPU know when the GPU is done rendering.

```
int num_pixels = nx*ny;
size_t fb_size = 3*num_pixels*sizeof(float);

// allocate FB
float *fb;
checkCudaErrors(cudaMallocManaged((void **)&fb, fb_size));
```

This call renders the image on the GPU and has the CUDA runtime divide the work on the GPU into blocks of 8×8 threads. You might also try other thread sizes to what works best for your machine. I tried to find a block size that:
(1) is a small, square region so the work would be similar. This should help each pixel do a similar amount of work. If some pixels work much longer than other pixels in a block, the efficiency of that block is impacted.
(2) has a pixel count that is a multiple of 32 in order to fit into warps evenly.

```
int tx = 8;
int ty = 8;
...
// Render our buffer
dim3 blocks(nx/tx+1,ny/ty+1);
dim3 threads(tx,ty);
render<<<blocks, threads>>>(fb, nx, ny);
checkCudaErrors(cudaGetLastError());
checkCudaErrors(cudaDeviceSynchronize());
```

Because of the way the book's code is written, I didn't choose to convert the final float values to 8-bit components on the GPU. As a further performance improvement, you could also choose to have the GPU convert the final float values to 8-bit components, saving bandwidth when the data is read back.

Threads launch in blocks and each block has 64 threads running the kernel function `render` (note the `__global__ identifier`). Using the `threadIdx` and `blockIdx` CUDA built-in variables we identify the coordinates of each thread in the image `(i,j)` so it knows how to calculate the final color. It is possible with images of sizes that are not multiples of the block size to have extra threads running that are outside the image. We must make sure these threads do not try to write to the frame buffer and return early.

Aside from that, the image calculation is very similar to the standard C++ code.

```
__global__ void render(float *fb, int max_x, int max_y) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if((i >= max_x) || (j >= max_y)) return;
    int pixel_index = j*max_x*3 + i*3;
    fb[pixel_index + 0] = float(i) / max_x;
    fb[pixel_index + 1] = float(j) / max_y;
    fb[pixel_index + 2] = 0.2;
}
```

Back on the host, after the `cudaDeviceSynchronize` we can access the frame buffer on the CPU host to output the image to stdout.

```
// Output FB as Image
std::cout << "P3\n" << nx << " " << ny << "\n255\n";
for (int j = ny-1; j >= 0; j--) {
    for (int i = 0; i < nx; i++) {
        size_t pixel_index = j*3*nx + i*3;
        float r = fb[pixel_index + 0];
        float g = fb[pixel_index + 1];
        float b = fb[pixel_index + 2];
        int ir = int(255.99*r);
        int ig = int(255.99*g);
        int ib = int(255.99*b);
        std::cout << ir << " " << ig << " " << ib << "\n";
    }
}
checkCudaErrors(cudaFree(fb));
```

We also `cudaFree` the frame buffer when we are finished with it. You can compare your code with mine here. With that you should produce a very similar result to your C++ program, with an example shown in figure 1. On Linux, there are many ways to view PPM images including the default viewer on Ubuntu (eog) which can view the PPM text output: `eog out.ppm`.

Figure 1. The first render

Adding Vectors

Chapter 2 in *Ray Tracing in One Weekend* introduce a vector class that can be used on both the CPU & GPU. To allow this, we annotate all of the vec3 class member functions with `__host__ __device__` keywords so we can execute them on the GPU and CPU.

```
class vec3  {
public:
    __host__ __device__ vec3() {}
    __host__ __device__ vec3(float e0, float e1, float e2) { e[0] = e0; e[1] = e1; e[2] = e2; }
    __host__ __device__ inline float x() const { return e[0]; }
    __host__ __device__ inline float y() const { return e[1]; }
    __host__ __device__ inline float z() const { return e[2]; }
...
```

Use these `__host__ __device__` keywords before all methods except the `operator>>` and `operator<<` methods.

Given this class, you can update your main.cu code to access an FB made up of `vec3` instead of 3 floats, similar to the C++ code. The end result shown in figure 2 should still look like figure 1. See the Chapter 2 reference implementation if you have any troubles.

Figure 2. Not much change in output, but the code is evolving

Classing Up the GPU

The rest of the C++ classes introduced in the book only get used on the GPU, so I annotate them with just `__device__` keywords. In Chapter 3, the book introduces the `ray` class.

```
class ray
{
  public:
    __device__ ray() {}
    __device__ ray(const vec3&amp; a, const vec3&amp; b) { A = a; B = b; }
    __device__ vec3 origin() const       { return A; }
    __device__ vec3 direction() const    { return B; }
    __device__ vec3 point_at_parameter(float t) const { return A + t*B; }

    vec3 A;
    vec3 B;
};
```

You will also need to add the `__device__` keyword to the `color` function since that is called from the render kernel.

We need to be careful to check for double precision code. Current GPUs run fastest when they do calculations in single precision. Double precision calculations can be several times slower on some GPUs. For example, this may look like a single precision calculation:

`float t = 0.5*(unit_direction.y() + 1.0);`

But, this code will calculate the multiplication and addition at double precision. While the `unit_direction.y()` result is float, since the floating-point constants 0.5 and 1.0 are doubles the float value is upconverted to double for the math. Only when the result is written into t will it be converted to single precision. To force single precision math, you need to coerce those constants to float. I typically add an "f" as a suffix. Note that the `vec3()` constructors coerce the values to float without any calculations, so you don't need to change those.

```
__device__ vec3 color(const ray& r) {
  vec3 unit_direction = unit_vector(r.direction());
  float t = 0.5f*(unit_direction.y() + 1.0f);
  return (1.0f-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
}

__global__ void render(vec3 *fb, int max_x, int max_y, vec3 lower_left_corner, vec3 horizontal, vec3 vertical, vec3 origin) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  if((i >= max_x) || (j >= max_y)) return;
  int pixel_index = j*max_x + i;
  float u = float(i) / float(max_x);
  float v = float(j) / float(max_y);
  ray r(origin, lower_left_corner + u*horizontal + v*vertical);
  fb[pixel_index] = color(r);
}
```

It is very easy to miss a double precision constant. For the most accurate test, use `nvprof`. For example:

`nvprof --metrics inst_fp_32,inst_fp_64 ./cudart > out.ppm`

This will output a report similar to the following:

```
Rendering a 200x100 image in 8x8 blocks.
==21852== NVPROF is profiling process 21852, command: ./cudart
took 0.103823 seconds.
==21852== Profiling application: ./cudart
==21852== Profiling result:
==21852== Metric result:
Invocations                 Metric Name                 Metric Description         Min        Max        Avg
Device "GeForce GTX 1070 (0)"
Kernel: render(vec3*, int, int, vec3, vec3, vec3, vec3)
1                  inst_fp_32               FP Instructions(Single)    1577600    1577600    1577600
1                  inst_fp_64               FP Instructions(Double)          0          0          0
```

This shows many single precision (`inst_fp_32`) and zero double precision (`inst_fp_64`) instructions. If you see any double precision instructions running on the GPU, you will not achieve the best performance.

Note that when running `nvprof`, you can temporarily change the size of your output image to make it smaller. This should make `nvprof` run very fast.

My code for this section can be found here.

Hit the Sphere

Chapter 4 uses concepts we've introduced above. There is a `hit_sphere` routine that is called from the GPU, so it requires a `__device__` annotation. In it, there are floating-point constants and math to attend to, too. The output should resemble figure 3.

See the Chapter 4 reference implementation if you have any troubles.

Figure 3. The first sphere

Manage Your Memory

The new topic introduces how we handle memory management for CUDA C++ classes on CPU & GPU. We are going to allocate our scene's objects on the GPU and this requires two pieces of code. First, we allocate a `d_list` of hitables and a `d_world` to hold those objects in the main routine on the CPU. (The `d_` prefix is for device-only data.) Next, we call a `create_world` kernel to construct those objects.

```
    hitable **d_list;
checkCudaErrors(cudaMalloc((void **)&amp;d_list, 2*sizeof(hitable *)));
   hitable **d_world;
   checkCudaErrors(cudaMalloc((void **)&amp;d_world, sizeof(hitable *)));
   create_world&lt;&lt;&lt;1,1&gt;&gt;&gt;(d_list,d_world);
   checkCudaErrors(cudaGetLastError());
   checkCudaErrors(cudaDeviceSynchronize());
```

In the kernel we make sure to only execute the construction of the world once

```
__global__ void create_world(hitable **d_list, hitable **d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list)   = new sphere(vec3(0,0,-1), 0.5);
        *(d_list+1) = new sphere(vec3(0,-100.5,-1), 100);
        *d_world    = new hitable_list(d_list,2);
    }
}
```

To use this world, we pass it to our `render` kernel and `color` function, along with the values describing our camera

```
render<<<blocks, threads>>>(fb, nx, ny,
                            vec3(-2.0, -1.0, -1.0),
                            vec3(4.0, 0.0, 0.0),
                            vec3(0.0, 2.0, 0.0),
                            vec3(0.0, 0.0, 0.0),
                            d_world);
```

And the kernels...

```
__global__ void render(vec3 *fb, int max_x, int max_y,vec3 lower_left_corner, vec3 horizontal, vec3 vertical, vec3 origin, hit
able **world)
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int j = threadIdx.y + blockIdx.y * blockDim.y;
   if((i >= max_x) || (j >= max_y)) return;
   int pixel_index = j*max_x + i;
   float u = float(i) / float(max_x);
   float v = float(j) / float(max_y);
   ray r(origin, lower_left_corner + u*horizontal + v*vertical);
   fb[pixel_index] = color(r, world);
}

__device__ vec3 color(const ray& r, hitable **world) {
   hit_record rec;
   if ((*world)->hit(r, 0.0, FLT_MAX, rec)) {
      return 0.5f*vec3(rec.normal.x()+1.0f, rec.normal.y()+1.0f, rec.normal.z()+1.0f);
   }
   else {
      vec3 unit_direction = unit_vector(r.direction());
      float t = 0.5f*(unit_direction.y() + 1.0f);
      return (1.0f-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
   }
}
```

After we're done, what we construct, we must delete; at the end of main you can see:

```
checkCudaErrors(cudaDeviceSynchronize());
free_world<<<1,1>>>(d_list,d_world);
checkCudaErrors(cudaGetLastError());
checkCudaErrors(cudaFree(d_list));
checkCudaErrors(cudaFree(d_world));
checkCudaErrors(cudaFree(fb));
```

And the `free_world` kernel is just:

```
__global__ void free_world(hitable **d_list, hitable **d_world) {
   delete *(d_list);
   delete *(d_list+1);
   delete *d_world;
}
```

In addition, there are `hitable.h`, `hitable_list.h`, and `sphere.h` files which all have classes and methods that are called from the GPU, so they require `__device__` annotations. Finally, there are floating-point constants and math to attend to, too. Figure 4 shows what your code should be generating.

My code for this section lies here.

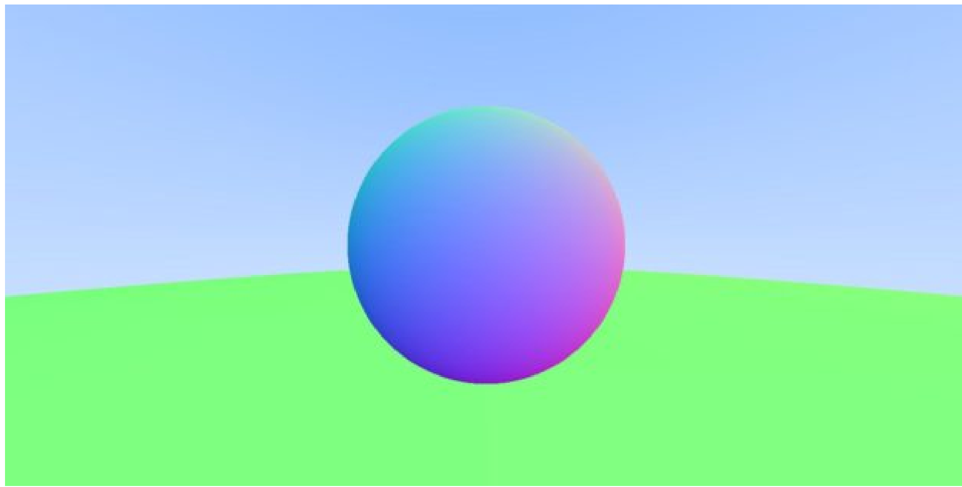


Figure 4. As images get more complex, we need to carefully manage memory on the GPU

It's All Random

Chapter 6 introduces stochastic or randomly determined values. Random numbers are a special topic for CUDA and requires the cuRAND library. Since "random numbers" on a computer actually consist of pseudorandom sequences, we need to setup and remember state for every thread on the GPU.

To do that we create a `d_rand_state` object for every pixel in our `main` routine.

```
#include <curand_kernel.h>
...
  curandState *d_rand_state;
  checkCudaErrors(cudaMalloc((void **)&d_rand_state, num_pixels*sizeof(curandState)));
...
  render_init<<<blocks, threads>>>(nx, ny, d_rand_state);
  checkCudaErrors(cudaGetLastError());
  checkCudaErrors(cudaDeviceSynchronize());
  render<<<blocks, threads>>>(fb, nx, ny,  ns, d_camera, d_world, d_rand_state);
  checkCudaErrors(cudaGetLastError());
  checkCudaErrors(cudaDeviceSynchronize());
```

We initialize `rand_state` for every thread. For performance measurement, I wanted to be able to separate the time for random initialization from the time it takes to do the rendering, so I made this a separate kernel. You might also choose to do this initialization at the start of your render kernel.

```
__global__ void render_init(int max_x, int max_y, curandState *rand_state) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  if((i >= max_x) || (j >= max_y)) return;
  int pixel_index = j*max_x + i;
  //Each thread gets same seed, a different sequence number, no offset
  curand_init(1984, pixel_index, 0, &rand_state[pixel_index]);
}
```

And finally, we pass it into our render kernel, make a local copy and use it per-thread. Replacing any `drand48()` calls with `curand_uniform(&local_rand_state)`.

```
__global__ void render(vec3 *fb, int max_x, int max_y, int ns, camera **cam, hitable **world, curandState *rand_state) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  if((i >= max_x) || (j >= max_y)) return;
  int pixel_index = j*max_x + i;
  curandState local_rand_state = rand_state[pixel_index];
  vec3 col(0,0,0);
  for(int s=0; s < ns; s++) {
    float u = float(i + curand_uniform(&local_rand_state)) / float(max_x);
    float v = float(j + curand_uniform(&local_rand_state)) / float(max_y);
    ray r = (*cam)->get_ray(u,v);
    col += color(r, world);
  }
  fb[pixel_index] = col/float(ns);
}
```

Note that now using debug flags in compilation makes a big difference in runtime. If you have those flags set on your compilation command line, remove them for a significant speedup. The image should resemble that shown in figure 4, earlier, but run faster.

My code for this chapter is here.

Iteration vs. Recursion

The default translation of the `color` function results in a stack overflow since it can call itself many times. But, the code can be simply be translated into a loop instead of recursion. In fact, later code in the book puts a limit of 50 levels of recursion, so we're just adding that limit a few chapters early.

The original C++ code looks like this and you can see if the `world->hit` returns true, the function recurs.

```
vec3 color(const ray& r, hitable *world) {
  hit_record rec;
  if (world->hit(r, 0.001, MAXFLOAT, rec)) {
    vec3 target = rec.p + rec.normal + random_in_unit_sphere();
    return 0.5*color( ray(rec.p, target-rec.p), world);
  }
  else {
    vec3 unit_direction = unit_vector(r.direction());
    float t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
  }
}
```

And I translated that into the following for CUDA:

```
__device__ vec3 color(const ray& r, hitable **world, curandState *local_rand_state) {
  ray cur_ray = r;
  float cur_attenuation = 1.0f;
  for(int i = 0; i < 50; i++) {
    hit_record rec;
    if ((*world)->hit(cur_ray, 0.001f, FLT_MAX, rec)) {
      vec3 target = rec.p + rec.normal + random_in_unit_sphere(local_rand_state);
      cur_attenuation *= 0.5f;
      cur_ray = ray(rec.p, target-rec.p);
    }
    else {
        vec3 unit_direction = unit_vector(cur_ray.direction());
        float t = 0.5f*(unit_direction.y() + 1.0f);
        vec3 c = (1.0f-t)*vec3(1.0, 1.0, 1.0) + t*vec3(0.5, 0.7, 1.0);
        return cur_attenuation * c;
      }
    }
  return vec3(0.0,0.0,0.0); // exceeded recursion
}
```

My code for this chapter is here.

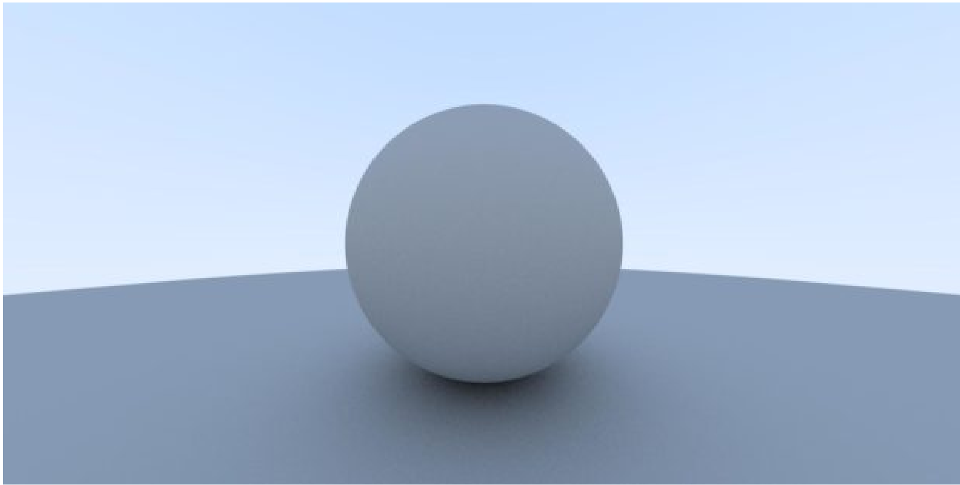


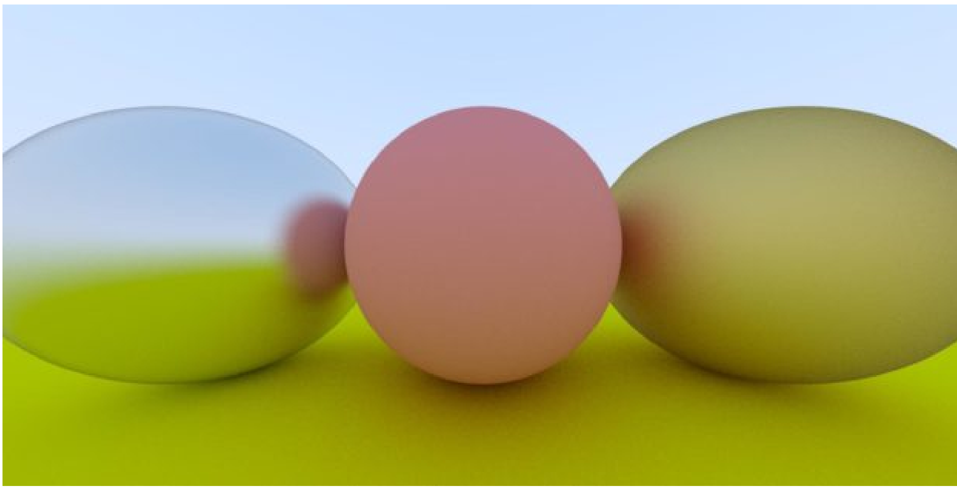
Figure 5. Now we see shadows

The Rest of the Chapters

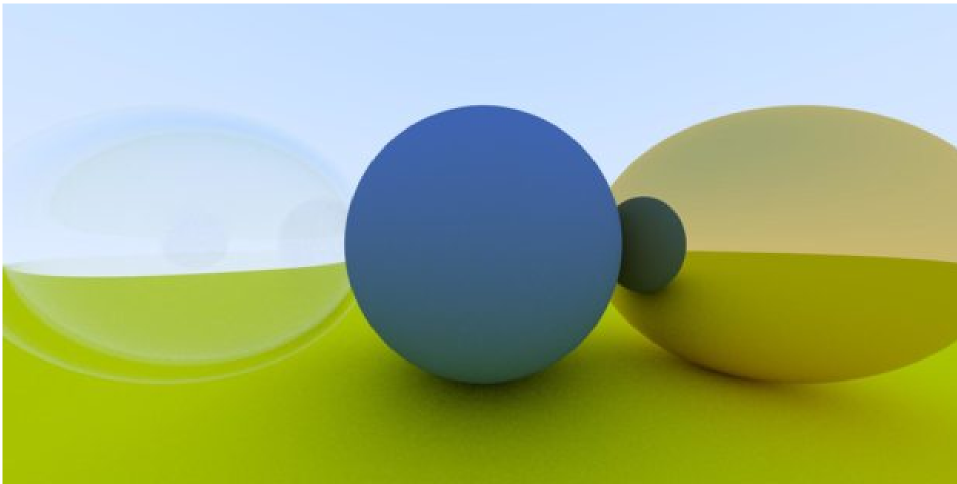
Chapters 8-12 use the same concepts we went over above to create the final image. You've learned all the concepts you need to finish the book. You should be able to take your C++ code, add the appropriate `__device__` annotations, add appropriate `delete` or `cudaFree` calls, adjust any floating point constants and plumb the local random state as needed to complete the translation. The images that follow show what your code should generate assuming you convert your code to CUDA correctly.

See the reference implementation code if you have any troubles.

Here is my code for Chapter 8.

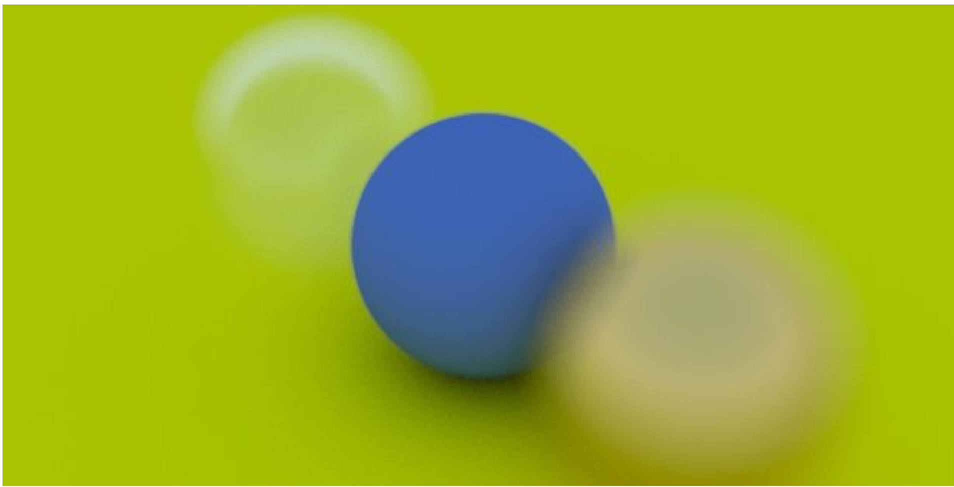


Here is my code for Chapter 9.

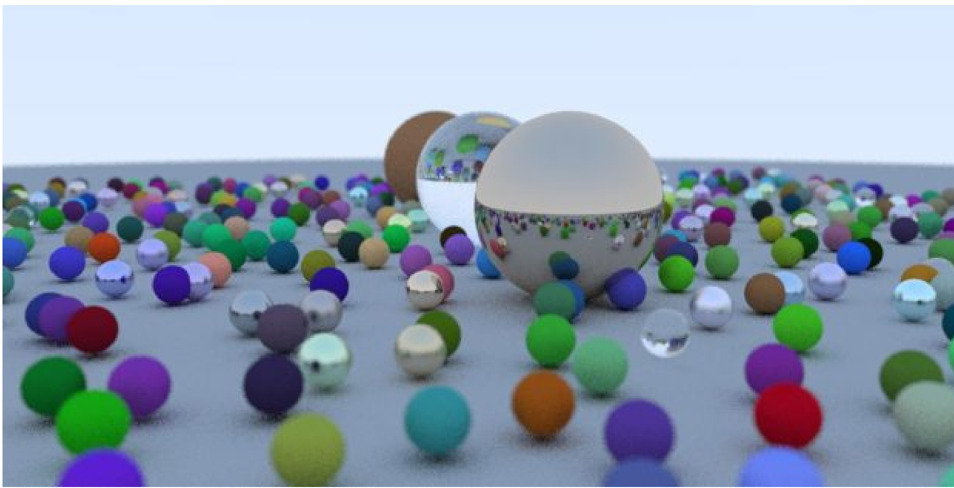


Here is my code for Chapter 10.

Here is my code for Chapter 11.



And finally, here is my code for the final chapter, Chapter 12. At this point, I hope you take a moment to compare the speedup from C++ to CUDA. My personal machine with a 6-core i7 takes about 90 seconds to render the C++ image. To give some concrete examples for the speedup you might see, on a Geforce GTX 1070, this runs in *6.7 seconds* for a 13x speedup. Using a new GeForce RTX 2080, the image renders in just *4.7 seconds* for a 19x speedup! What speedup do you see? When you try on your machine, be sure to try a range of thread block sizes (`tx` and `ty` in `main.cu`) to see if that helps your performance.



Next

I hope you enjoyed this walkthrough and it helps you in the process of creating a CUDA ray tracer. This is only a simple and relatively direct translation from C++ to CUDA. We didn't try any further acceleration techniques and yet achieved up to a 19x speedup. This is just the start on a journey you might enjoy for a long time–ray tracing is a deeply interesting subject. Here are a few more ideas to consider…

- continue reading the rest of the Peter Shirley's series: Ray Tracing: The Next Week, and Ray Tracing: The Rest of Your Life.
- look into using the OptiX API which uses CUDA as the shading language, has CUDA interoperability and accesses the latest Turing RT Cores for hardware acceleration.
- Microsoft has announced DirectX 3D Ray Tracing, and NVIDIA has announced new hardware to take advantage of it–so perhaps now might be a time to look at real-time ray tracing?
- do you have multiple GPUs? Consider adding even more parallelism by using them!

Acknowledgements

I would like to thank Mark Harris, Peter Shirley, Pawel Tabaszewski and Loyd Case for their helpful insights and encouragement in creating this post. I would also like to thank github user pfranz for making the first github repo with a tag for each chapter which was very handy.

About the Authors

About Roger Allen

Ce site utilise des cookies provenant de Google pour fournir ses services et analyser le trafic. Votre adresse IP et votre user-agent, ainsi que des statistiques relatives aux performances et à la sécurité, sont transmis à Google afin d'assurer un service de qualité, de générer des statistiques d'utilisation, et de détecter et de résoudre les problèmes d'abus.

EN SAVOIR PLUS OK

in One Weekend

Weekends are a great chunk of time to get a significant project done, and limiting it to a weekend helps to stay on task. This blog is inspired by a how-to book on ray tracing, but that is just the initial driving example.

Wednesday, January 27, 2016

Ray Tracing in One Weekend

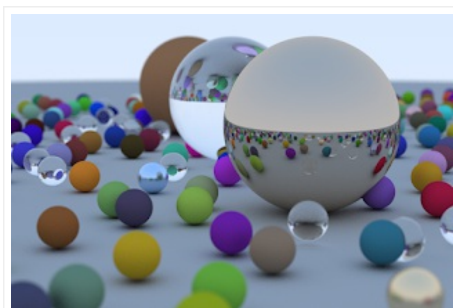
The books are now available in unprotected pdf for PAY WHAT YOU WISH, with 50% going to not for profit programming education organizations. Here are the files.

Ray tracing was invented by Turner Whitted around 1980. His classic paper is really amazing; it had bounding volume hierarchies, adaptive sampling, and it's future work suggested randomized reflection. Since then it has branched into a ton of variations and is used in most movies for the indirect "bounce" lighting.

I've was assigned a 2D ray tracer in a physics class in 1984 and I was hooked. Since then I have written a bunch of ray tracers in Pascal, Fortran, Scheme, C, C++, Java, and most recently Swift (which is awesome for writing ray tracers). I've also taught classes with ray tracing about a dozen times, most recently at Westminster College in 2015. My approach over the years has evolved to do what I think are the most fun parts of ray tracing that are still in the direction of writing a production-quality ray tracer.

Ray Tracing in One Weekend is a kindle book that goes through all of the details to generate a rudimentary ray tracer. It's $2.99 on amazon. It uses C plus classes plus operator overloading. I have heard this referred to as "C plus" which I now call it. Most production renderers are written in C++ so I opted for that as the driving language. I put the most primal set of these assignments so that it's very doable in a weekend, the fundamental unit of coding self-improvement :) Here is the book:

The resulting program generates this image:



Here's my MacOS code for the book. Should be semi-portable but drand48() will need to be written on some systems and the fileio may be too. I also have a little post on the tools I used to write the book for those interested in doing a little book of their own. Please let me know if you do!

About Me

View my complete profile

Blog Archive

▼ 2016 (4)
► March (2)
▼ January (2)
Ray Tracing: the Next Week
Ray Tracing in One Weekend

Links for further reading and exploration related to the book:

Chapter 0: Overview

Here is a list of basic graphics texts that cover the vector background the book assumes:

Fundamentals of Computer Graphics, Fourth Edition

Computer Graphics: Principles and Practice (3rd Edition)

The Graphics Codex

Real-Time Rendering, Third Edition

Chapter 1: Output an image

For output of most LDR and HDR images I lovestb_image.

For more info on HDR images and tone mappingthe I like the book **High Dynamic Range Imaging, Second Edition: Acquisition, Display, and Image-Based Lighting**

Chapter 2: The vec3 class

I advocate using the same class for points, displacements, colors, etc. Some people like more structure and type checking (so for example multiplying two locations would be rejected by the compiler). An example article where points and vectors are different is here. Jim Arvo and Brian Smits experimented with not only distinguishing points and vectors, but using the compiler to do dimensional analysis (so velocity, length, and time would be different types for example). They found this to be too cumbersome in 1990s C++ but I'd love to hear about anybody's experience. Researchers at Dartmouth have taken a really serious effort at this and their code and paper are available at github.

Chapter 3: Rays, a simple camera, and background

The first thing you might add to your background function is an environment map. Paul Debevec has a terrific history of environment mapping. The easiest mapping for this uses a single image for the entire sphere of directions. Paul also provides some good images to use for your environment map.

Chapter 4: Adding a sphere

There are a bunch of other object types you can add. Triangles are usually first and I am a fan of barycentric methods. After triangles, many people quit adding primitives because graphics has such a big infrastructure for triangles. Ellipses are an easy thing to add but instancing is usually a more "ray tracey" approach (let the software do the heavy lifting). Composite objects via CSG are surprisingly straightforward.

Chapter 5: Surface normals and multiple objects.

If you want your code to be more efficient for large numbers of objects, use a BVH-- they are as good as any other in efficiency and are the most robust and easiest to implement.

Chapter 6: Antialiasing

Box filtering as done in the book suffices for most situations. However, Guassian-like filters can have advantages and are pretty easy. You can either uniformly sample the whole screen and weight the samples, or use non-uniform samples. All approaches work.

Chapter 7: Diffuse Materials

"Ideal" diffuse materials, also called "Lambertian" are used 99% of the time in graphics. The wikipedia article on this approximation is good. Real diffuse surfaces do not behave exactly as Lambertian (for example they get specular at grazing angle) but especially with interreflection in the mix the appearance differences are minor. So this is probably not where you should push your renderer until many many other features are addressed.

Chapter 8: Metal

The first improvement you might make is to have the color of the metal go to white at grazing angle. The Schlick approximation (used in Chapter 9 for glass where grazing behavior matters more) works for that. Full-bore Fresnel equations will describe color variation with angle, but in my experience getting normal incident color is good enough.

Chapter 9: Dielectrics

The first thing you might add is filtering of light within the dielectric (like the subtle "greenness" of much glass). This is a classic exponential decay and is covered well in the Beer's Law section of this nice page.

Chapter 10: Positionable camera

Camera parameter setting is just plain ugly. The system used in the book is relatively common and is in my opinion the prettiest. I avoid the matrix formulation wherever possible because I never understand my code when I am done. But it is very elegant and works.

Chapter 11: Defocus blur

If you ever need to match a real camera, there is a nice survey of models used in graphics. And here is a very good presentation on real lenses written for a technical audience.

Chapter 12: Where next?

You should have the core of a very serious ray tracer now. I would now take it in one of three directions. They are not mutually exclusive but explicitly deciding your goals will simplify architectural decisions.

1. Make it physically accurate. This will imply using spectra instead of RGB (I like just using a big array of wavelengths) and get something where you know the reflectances. Popular is to get a X-Rite MSCCC ColorChecker Classic whose data is available online.
2. Make it good for generating animations. Lots of movies use a ray traced system, and Disney, Pixar, and the Solid Angle teams have both disclosed a remarkable amount about their code. Work on features and then efficiency. I think you will be amazed how soon you can produce amazing images.
3. Make it fast. Here you can roll your own, or start using a commercial API. To see exactly where that community is now, go to the 2016 HPG conference. Or backtrack in their previous papers. They are a very open community and the papers go into much detail relative to many sub-fields in computer graphics.

Please send me your questions, comments, and pictures. Have fun with ray tracing!

Posted by Peter Shirley at 2:34 PM

106 comments:

Michael Mellinger January 27, 2016 at 6:59 PM

You should consider publishing in iBooks too because it's easier to push updates. I imagine the book could be perfect but most computer books seem to have a few mistakes. It's just the nature of the topic.

Reply

Replies

Peter Shirley February 4, 2016 at 9:06 AM

Thanks-- I already pushed an update so no not perfect! I'll look into iBooks. I think currently I have enabled some amazon features related to loaning and kindle unlimited free reading. I'll do some due diligence on the ecosystem once I finish part 2. I've had requests for pdf as well and there are a bunch of options there I don't yet understand...

Reply

Tzu-Mao Li February 3, 2016 at 11:38 AM

SafeGI is a good example of the usage of dimensional analysis in a rendering system. https://github.com/ouj/safegi

Reply

Replies

Peter Shirley February 4, 2016 at 9:04 AM

Really interesting work. Thanks for the link. The github has the EGSR paper as well. I'll