School of Data and Computer Science, Sun Yat-sen University

CUDA 和 BVH

18308045 Zhengyang Gu

January 2, 2021

Contents

1 Abstract

基础版本的光线追踪的大致流程是：将图像的每个像素视作射入摄像头的每一束光线，串行地计算每个射入光线的颜色；计算射入光线时需要遍历全部的物体以获得光线最近接触物体的一些信息来进行后续计算。这两个点就分别是 CUDA 和 BVH 所要优化的地方。

2 CUDA

由于每个射入光线是没有依赖关系的，所以理想情况下可以对每个射入光线并行计算。在这里我用 CUDA 来实现射入光线间的并行。

2.1 实现要点

2.1.1 写图像

写图像对每个像素点是有顺序要求的，要求从左上角一行行写到右下角。而最初的实现是将计算射入光线颜色和写像素点两个步骤合在一起的，即每算完一个射入光线就会写一个像素点：

```
// Render

std::cout << "P3\n" << image_width << "␣" << image_height << "\n255\n";

for (int j = image_height-1; j >= 0; --j) {
    std::cerr << "\rScanlines␣remaining:␣" << j << '␣' << std::flush;
    for (int i = 0; i < image_width; ++i) {
        color pixel_color(0, 0, 0);
        for (int s = 0; s < samples_per_pixel; ++s) {
            auto u = (i + random_double()) / (image_width-1);
            auto v = (j + random_double()) / (image_height-1);
            ray r = cam.get_ray(u, v);
            pixel_color += ray_color(r, world, max_depth);
        }
        write_color(std::cout, pixel_color, samples_per_pixel);
    }
}
```

write_color 即是将像素点写入流。这样不利于并行。这里我将这两个步骤分开操作，先并行地算完所有像素点的颜色，再串行地写入图像：

```
// Output FB as Image
std::cout << "P3\n" << image_width << '␣' << image_height << "\n255\n";
```

```
for (int j = image_height - 1; j >= 0; j--) {
    for (int i = 0; i < image_width; i++) {
        size_t pixel_index = j * 3 * image_width + i * 3;
        std::cout << fb[pixel_index + 0] << '' << fb[pixel_index + 1] << '' << fb[
            pixel_index + 2] << '\n';
    }
}
```

其中 fb 是已经算好的全部像素点的 RGB。

2.1.2 浮点数精度

原先的实现中浮点数精度默认是双精度，如：

```
// Constants

const double infinity = std::numeric_limits<double>::infinity();
const double pi = 3.1415926535897932385;

// Utility Functions

inline double degrees_to_radians(double degrees) {
    return degrees * pi / 180.0;
}
```

其中有显性的双精度 double 和隐性的双精度 180.0。而现在的 GPU 在双精度的计算上比单精度慢数倍，因而为了进一步提速，将所有的双精度都改成单精度：

```
// Constants

__device__ float infinity = std::numeric_limits<float>::infinity();
__device__ float pi = 3.14159265358979323f;

// Utility Functions

__device__ inline float degrees_to_radians(float degrees) {
    return degrees * pi / 180.0f;
}
```

2.1.3 STL 的数据结构

原先的实现运用了很多 STL 的数据结构如 vector，shared_ptr：

```
class hittable_list : public hittable {
    public:
        hittable_list() {}
        hittable_list(shared_ptr<hittable> object) { add(object); }

        void clear() { objects.clear(); }
        void add(shared_ptr<hittable> object) { objects.push_back(object); }

        virtual bool hit(
            const ray& r, double t_min, double t_max, hit_record& rec) const override;

    public:
        std::vector<shared_ptr<hittable>> objects;
};
```

而由于 STL 的数据结构是基于内存而非显存实现的，无法被 GPU 利用。所以在这里都改成普通的指针：

```
class hittable_list : public hittable {
    public:
        __device__ hittable_list() {}
        __device__ hittable_list(hittable** list, size_t len) : objects(list), size(len)
            { }
        __device__ virtual bool hit(
            const ray& r, float t_min, float t_max, hit_record& rec) const override;
        __device__ virtual bool bounding_box(
            float time0, float time1, aabb& output_box) const override;

    public:
        hittable** objects;
        size_t size;
};
```

2.1.4 随机数

在原本的实现中诸多地方用到了随机数，如去锯齿、求漫反射模拟摄像头的光圈等等，这些随机数都用到了 cstdlib 的随机数实现：

```
#include <cstdlib>
...

```

```
inline double random_double() {
    // Returns a random real in [0,1).
    return rand() / (RAND_MAX + 1.0);
}
```

而这个库是 host 的函数不能被 device 调用。在 device 函数中的随机数，需要用 curand_kernel.h 库的实现：

```
#include <curand_kernel.h>
...

__device__ inline float random_float(curandState* local_rand_state) {
    // Returns a random real in [0,1).
    return curand_uniform(local_rand_state);
}
```

而在计算机上的随机数实现实际上是通过伪随机数的实现，因而需要记录 GPU 上每个线程的状态，这就需要开设并初始化一个 rand_state 空间：

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

2.1.5 数学函数

原先实现运用到的一些数学函数是来自 cmath 库的：

```
#include <cmath>
...

vec3 refract(const vec3& uv, const vec3& n, double etai_over_etat) {
    auto cos_theta = fmin(dot(-uv, n), 1.0);
    vec3 r_out_perp =  etai_over_etat * (uv + cos_theta*n);
    vec3 r_out_parallel = -sqrt(fabs(1.0 - r_out_perp.length_squared())) * n;
    return r_out_perp + r_out_parallel;
}
```

而这些函数同样是 host 函数，没法被 device 函数调用，所以这里单独建了个 math.h 头文件来自己实现一些用得到的数学函数：

```
#ifndef MATH_H
#define MATH_H

__device__ inline float fmin(float& a, float& b)
{
    return a <= b ? a : b;
}

__device__ inline float fmax(float& a, float& b)
{
    return a >= b ? a : b;
}

__device__ inline float fabs(float& a)
{
    return a >= 0 ? a : -a;
}

#endif
```

2.1.6 尾递归优化

原先实现中计算光线颜色的地方用到了不必要的尾递归：

```
color ray_color(const ray& r, const hittable& world, int depth) {
    hit_record rec;

    // If we've exceeded the ray bounce limit, no more light is gathered.
    if (depth <= 0)
        return color(0,0,0);

    if (world.hit(r, 0.001, infinity, rec)) {
        ray scattered;
        color attenuation;
        if (rec.mat_ptr->scatter(r, rec, attenuation, scattered))
            return attenuation * ray_color(scattered, world, depth-1);
        return color(0,0,0);
    }

```

```
    vec3 unit_direction = unit_vector(r.direction());
    auto t = 0.5*(unit_direction.y() + 1.0);
    return (1.0-t)*color(1.0, 1.0, 1.0) + t*color(0.5, 0.7, 1.0);
}
```

这可能会浪费栈内存导致栈溢出，因而改成循环：

```
__device__ color ray_color(const ray& r, hittable_list** world, int max_depth,
    curandState *local_rand_state) {
    ray cur_ray = r;
        color cur_attenuation = vec3(1.0f, 1.0f, 1.0f);
    for(int i = 0; i < max_depth; i++) {
        hit_record rec;
        if ((*world)->hit(cur_ray, 0.001f, infinity, rec)) {
                        ray scattered;
                        color attenuation;
                        if (rec.mat_ptr->scatter(cur_ray, rec, attenuation, scattered,
                            local_rand_state))
                        {
                                cur_attenuation = cur_attenuation * attenuation;
                                cur_ray = scattered;
                        }
                        else
                                return color(0,0,0);
        }
        else {
            vec3 unit_direction = unit_vector(cur_ray.direction());
                        float t = 0.5f * (unit_direction.y() + 1.0f);
                        vec3 c = (1.0f - t) * vec3(1.0f, 1.0f, 1.0f) + t * vec3(0.5, 0.7
                            f, 1.0f);
            return cur_attenuation * c;
        }
    }
        return vec3(0.0f, 0.0f, 0.0f); // exceeded recursion
}
```

2.2 实验

该部分实验基于 Ray Tracing In One Weekend 的最终场景，尝试了几组 block 的形状，每组分别测三次运行时间，并测一次原先 CPU 实现的运行时间作为对照。

2.2.1 场景设置

参数设置如下：

```
// Image
const auto aspect_ratio = 3.0 / 2.0;
const int image_width = 600;
const int image_height = static_cast<int>(image_width / aspect_ratio);
const int samples_per_pixel = 500;
const int max_depth = 50;
```

并将场景中的小球，只保留了 4 个大球。

2.2.2 block 形状确定

CUDA 的 device 实际在执行的时候，会以 Block 为单位，把一个个的 block 分配给 SM 进行运算；而 block 中的 thread，又会以「warp」为单位，把 thread 来做分组计算。目前 CUDA 的 warp 大小都是 32，也就是 32 个 thread 会被群组成一个 warp 来一起执行。在 Compute Capability 1.0/1.1 中，每个 SM 最多可以同时管理 768 个 thread（768 active threads）或 8 个 block（8 active blocks）。为了让工作均衡，这就要求了：

- 每个 block 大小是 32 的倍数，这样不同 warp 大小一致。
- block 的宽和高要分别整除图像的宽和高，这样不同 block 形状类似。

由于 image_width=600,image_height=400，我选择了块宽和块高：4x8,8x8,8x16,10x16,24x16,24x20。

2.2.3 结果

```
jovyan@jupyter-adva1g10:~/ray_tracing_cpu$ time ./main_cpu > test_cpu.ppm
Scanlines remaining: 0
Done.

real    3m2.126s
user    3m1.463s
sys     0m0.004s
```

Figure 1: CPU

```
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 4x8 blocks.
took 1.95163 seconds.

real    0m2.192s
user    0m2.062s
sys     0m0.113s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 4x8 blocks.
took 1.97175 seconds.

real    0m2.222s
user    0m2.074s
sys     0m0.124s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 4x8 blocks.
took 1.92809 seconds.

real    0m2.176s
user    0m2.030s
sys     0m0.128s
```

Figure 2: CUDA with block shape 4x8

```
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 8x8 blocks.
took 1.93453 seconds.

real    0m2.183s
user    0m2.044s
sys     0m0.117s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 8x8 blocks.
took 1.94878 seconds.

real    0m2.208s
user    0m2.033s
sys     0m0.148s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 8x8 blocks.
took 1.95481 seconds.

real    0m2.198s
user    0m2.050s
sys     0m0.132s
```

Figure 3: CUDA with block shape 8x8

```
Rendering a 600x400 image with 500 samples per pixel in 8x16 blocks.
took 1.93183 seconds.

real    0m2.180s
user    0m2.039s
sys     0m0.125s
jovyan@jupyter-adva1g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 8x16 blocks.
took 1.93452 seconds.

real    0m2.192s
user    0m2.035s
sys     0m0.128s
jovyan@jupyter-adva1g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 8x16 blocks.
took 1.92172 seconds.

real    0m2.188s
user    0m2.023s
sys     0m0.128s
```

Figure 4: CUDA with block shape 8x16

```
jovyan@jupyter-adva1g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 10x16 blocks.
took 2.00302 seconds.

real    0m2.273s
user    0m2.108s
sys     0m0.124s
jovyan@jupyter-adva1g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 10x16 blocks.
took 1.90062 seconds.

real    0m2.146s
user    0m1.980s
sys     0m0.148s
jovyan@jupyter-adva1g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 10x16 blocks.
took 1.92955 seconds.

real    0m2.184s
user    0m2.028s
sys     0m0.128s
```

Figure 5: CUDA with block shape 10x16

```
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.81039 seconds.

real    0m2.073s
user    0m1.923s
sys     0m0.116s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.85449 seconds.

real    0m2.100s
user    0m1.960s
sys     0m0.120s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.84003 seconds.

real    0m2.085s
user    0m1.951s
sys     0m0.120s
```

Figure 6: CUDA with block shape 24x16

```
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x20 blocks.
took 1.9339 seconds.

real    0m2.183s
user    0m2.037s
sys     0m0.124s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x20 blocks.
took 1.9201 seconds.

real    0m2.166s
user    0m2.009s
sys     0m0.140s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x20 blocks.
took 1.95492 seconds.

real    0m2.197s
user    0m2.046s
sys     0m0.136s
```

Figure 7: CUDA with block shape 24x20

可以看出加速比在 block size 为 24x16 时到达顶峰，接近 93，而 size 过大或过小都没有更优。

上面分析了引入 CUDA 效率，下面分析正确性。由于随机数因素，难以逐字节对比文件来分析，这里通过直观地对比图片来分析：

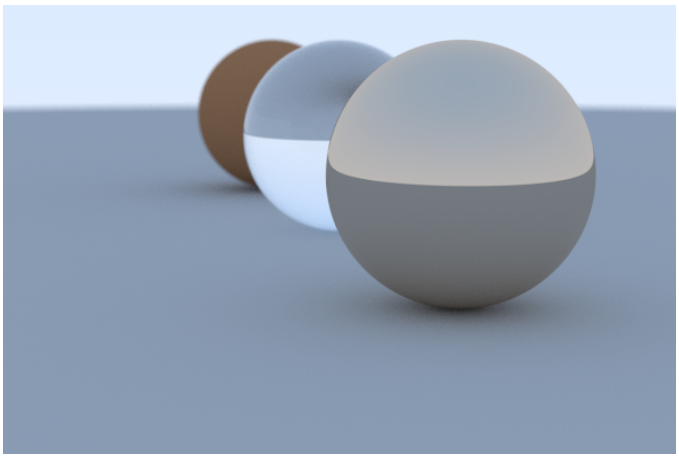

Figure 8: CPU version

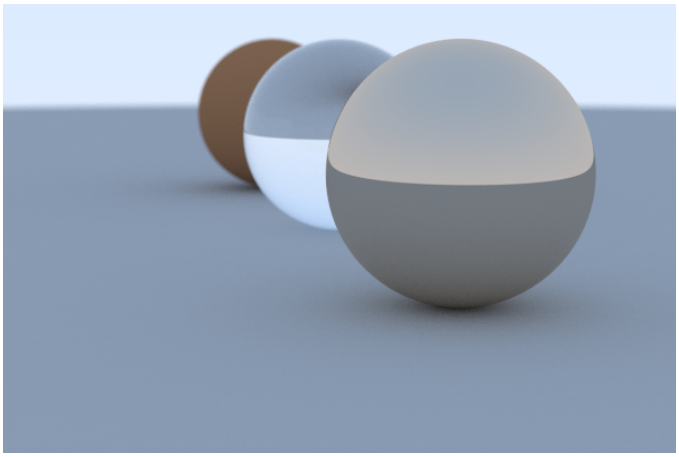

Figure 9: CUDA version

就算有随机因素的情况下，直观上仍找不出两图的差别，基本说明是正确的。

3 BVH

获取与最近接触物体的信息，可以通过 BVH 树搜索优化，来使其由线性复杂度变为对数复杂度。

3.1 原理简介

3.1.1 BVH

BVH 结构如下：

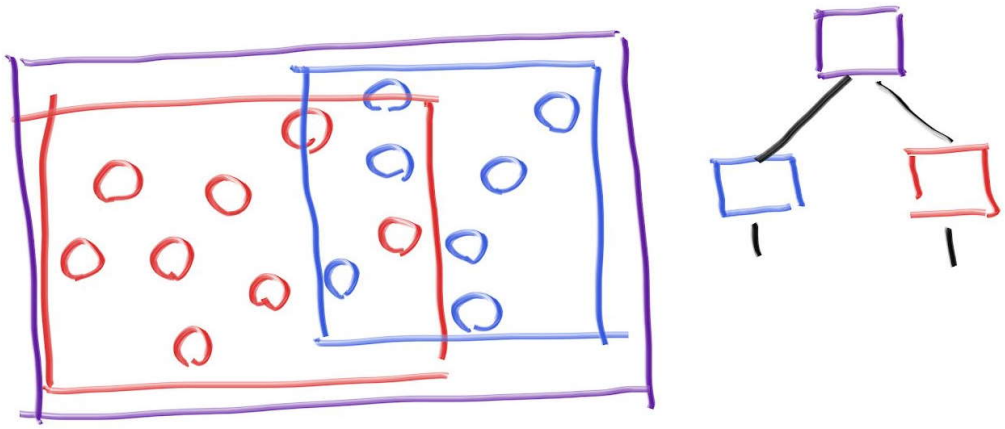

Figure 10: Bounding volume hierarchy

圆形是物体即叶子节点，方框是非叶子节点。每个父节点有两个子节点，图形上父节点是包含两个子节点的大框。这样假设物体数量为 N，如果就能构造出比较平衡的二叉树（即划分左右节点比较均衡），那它的树高最好可以达到 $\log(N)$。而找最近接触物体的伪代码如下：

```
if (hits purple)
    hit0 = hits blue enclosed objects
    if (hit0)
        return true and info of closer hit
    hit1 = hits red enclosed objects
    if (hit1)
        return true and info of closer hit
return false
```

由于可以通过 hits purple 和 hit0 来提前结束搜索，令其只需遍历一条根到叶子的路径，因而击中物体的情况下最好时间复杂度是 $O(\log(N))$。而也有可能击中了全部框体但是却只击中了二叉树中最后一个物体，但就算是这种最坏的情况，由于二叉树一共 $2N-1$ 个节点，其时间复杂度也是 $O(N)$。因而如果能构造出一个比较平衡的二叉树其时间复杂度上将线性优化成了对数。

3.1.2 AABB

上述所提到的 $O(N)$ 最坏情况是难以避免的。如果要构造出完全没缝隙的框体，这样会增加 hits purple 的判断复杂度。考虑两个完全不相交的球，没缝隙的框体是两块球空间的并集，而判断 hits purple 则还是只能分别判断是否击中两个球，这样效率只能更低。因此换句话说，构造一个简单的框体是很重要的。这里使用 Axis-Aligned Bounding Boxes，其实际上是用与轴平行的线划分方形空间。其计算 hits purple 是很简单的，只需看该空间的所有维度的区间在光线上的投影是否重叠：

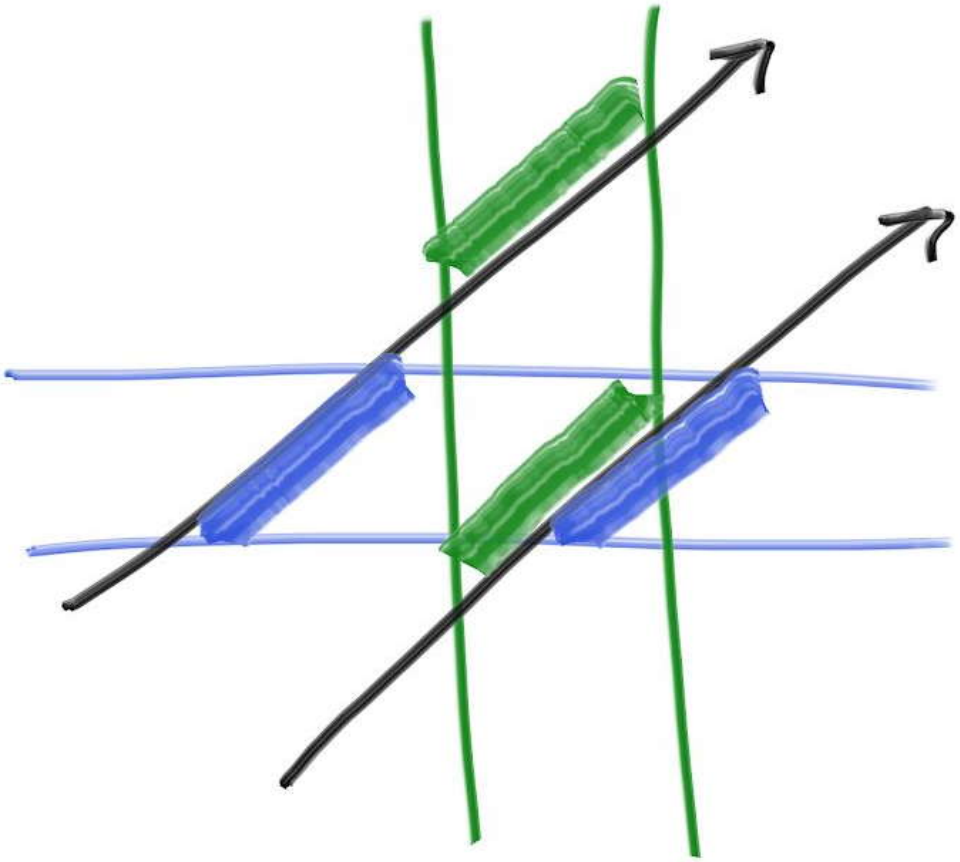

Figure 11: Ray-slab t-interval overlap

上图中，两条黑线表示两束光线，蓝色和绿色分别表示两个维度。两个维度在上面光线的投影不重合，因而上面光线不穿过中间的方形区间；两个维度在下面光线的投影重合，因而下面光线穿过方形区间。

3.1.3 构建 BVH

另一个问题是要达到对数时间复杂度，还要能够造出比较平衡的 BVH 树。这里使用的方法是每次划分时，随机取一个维度，在上面排序物体，然后二分令两部分物体数量一样。值得注意的是由于

CUDA，排序算法无法使用 STL 提供的排序，在这里我使用了 thrust 库提供的排序。

3.2 实验

该部分实验基于 Ray Tracing In One Weekend 的最终场景，CUDA block 形状使用最好的 24x16，由于这部分优化和物体数量有关，我设置了不同物体数量，分别为：4 大球、4 大球 484 小球、4 大球 1936 小球。

3.2.1 场景设置

4 大球的场景和 CUDA 实验相同，后面更多球的场景为了加快渲染时间将 num_samples 降到了 100。

3.2.2 更多球数量的实现

4 大球和 CUDA 实验相同，484 小球只是将注释掉的部分恢复，1936 小球是将 484 小球的每 1 个改成 4 个重叠的小球这样大量框体无空隙是很能体现 BVH 优势的。

3.2.3 结果

```
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.81039 seconds.

real    0m2.073s
user    0m1.923s
sys     0m0.116s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.85449 seconds.

real    0m2.100s
user    0m1.960s
sys     0m0.120s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel in 24x16 blocks.
took 1.84003 seconds.

real    0m2.085s
user    0m1.951s
sys     0m0.120s
```

Figure 12: 4 balls without BVH

```
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel with BVH in 24x16 blocks.
took 2.40105 seconds.

real    0m3.457s
user    0m3.254s
sys     0m0.156s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel with BVH in 24x16 blocks.
took 2.46697 seconds.

real    0m2.711s
user    0m2.551s
sys     0m0.144s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 500 samples per pixel with BVH in 24x16 blocks.
took 2.3935 seconds.

real    0m2.663s
user    0m2.479s
sys     0m0.144s
```

Figure 13: 4 balls with BVH

```
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test_no_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 9.38127 seconds.

real    0m10.520s
user    0m10.308s
sys     0m0.132s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test_no_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 9.63337 seconds.

real    0m9.912s
user    0m9.741s
sys     0m0.136s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test_no_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 9.29504 seconds.

real    0m9.615s
user    0m9.432s
sys     0m0.116s
```

Figure 14: 488 balls without BVH

```
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 10.8809 seconds.

real    0m11.226s
user    0m11.034s
sys     0m0.136s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 11.0644 seconds.

real    0m11.396s
user    0m11.225s
sys     0m0.136s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test_bvh.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 11.0528 seconds.

real    0m11.413s
user    0m11.210s
sys     0m0.140s
```

Figure 15: 488 balls with BVH

```
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 38.47 seconds.

real    0m39.898s
user    0m39.539s
sys     0m0.168s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 37.3123 seconds.

real    0m37.914s
user    0m37.609s
sys     0m0.160s
jovyan@jupyter-adval g10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel without BVH in 24x16 blocks.
took 39.8807 seconds.

real    0m40.479s
user    0m40.213s
sys     0m0.148s
```

Figure 16: 1940 balls without BVH

```
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 13.5215 seconds.

real    0m16.158s
user    0m15.240s
sys     0m0.156s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 13.5362 seconds.

real    0m14.649s
user    0m14.444s
sys     0m0.137s
jovyan@jupyter-advalg10:~/ray_tracing$ time ./main > test.ppm
Rendering a 600x400 image with 100 samples per pixel with BVH in 24x16 blocks.
took 13.2302 seconds.

real    0m14.358s
user    0m14.141s
sys     0m0.132s
```

Figure 17: 1940 balls with BVH

可以看出在大量物体且缝隙少的场景下，BVH 的加速效果明显。而在少量物体场景下，BVH 反而慢，因为 CUDA 执行流复杂任务效果不佳，另外 BVH 需要大量常数时间。由于 BVH 的优化只和物体数目有关，在渲染前可以调小其他参数看开启 BVH 有无优化再决定是否开启 BVH。

下面是正确性的验证，两张图都是从 488 球的场景下生成的：

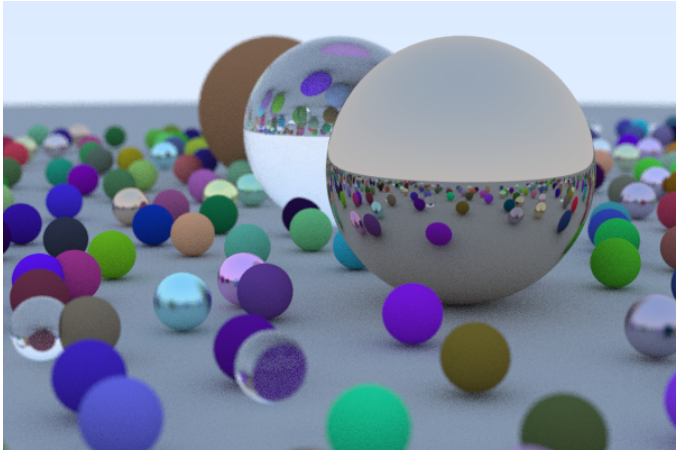

Figure 18: image without BVH

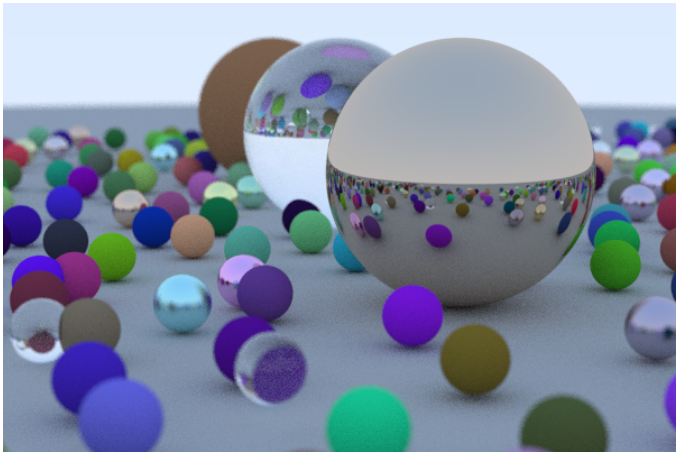

Figure 19: image with BVH

就算有随机因素的情况下，直观上仍找不出两图的差别，基本说明是正确的。