Parallelized Ray Tracing In One Weekend

Yao-Te Ying*
leaf.ying.cs11@nycu.edu.tw
Institute of Multimedia Engineering in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

Ying-Ting Lai*
ytlai.cs12@nycu.edu.tw
Institute of Computer Science and Engineering in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

Chun-Hung Wu*
wu891123.cs09@nycu.edu.tw
Department of Computer Science in National Yang Ming Chiao Tung University
Hsinchu, R.O.C.

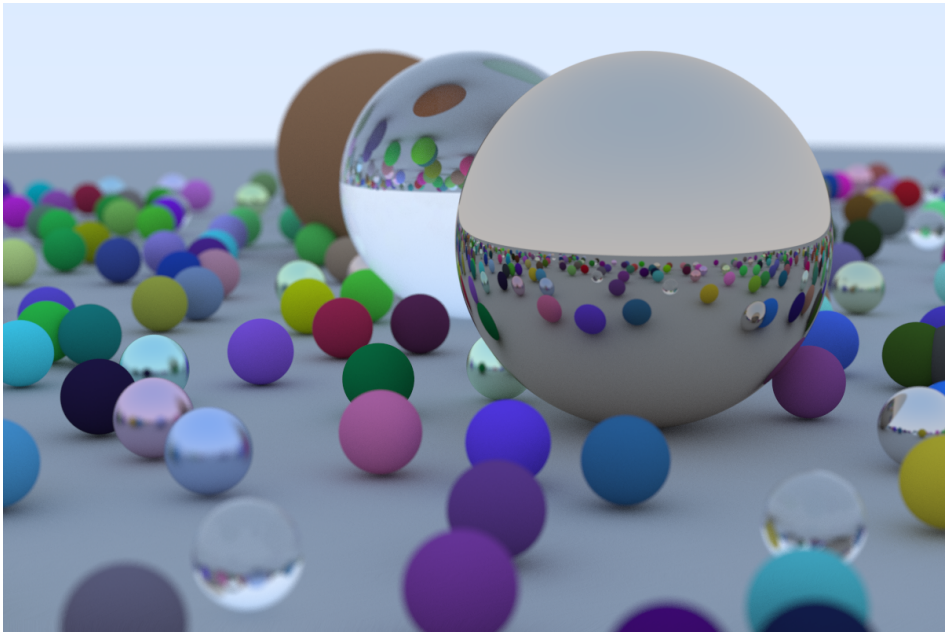


Figure 1: Final scene

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**.

KEYWORDS

CUDA, OpenMP, Ray Tracing, Computer Graphics, C++

1 ABSTRACT

In this project, we parallelized the first book of the Ray Tracing In One Weekend series [3], uncovering the parallelizable nature of ray tracing. We explored performance across hardware levels, showcasing multi-threading scalability with OpenMP on CPU cores. Additionally, we optimized GPU acceleration with CUDA. The CPU parallelization yielded a 5x speedup over the serial program, with load-balanced processing 1.25x faster than imbalanced. GPU acceleration achieved an impressive 25x speedup. The source code of this research project is available at https://github.com/guyleaf/Parallelized-Ray-Tracing-In-One-Weekend.

*All authors contributed equally to this research.

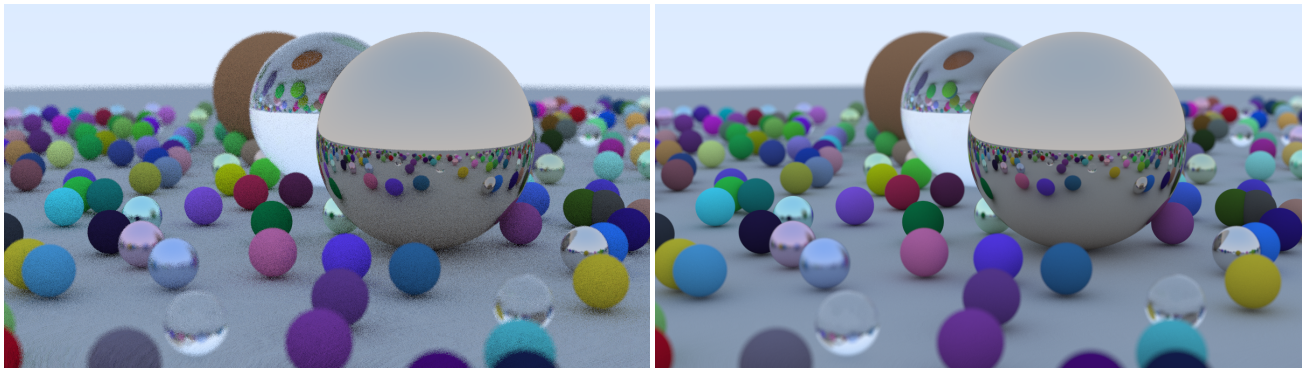
2 INTRODUCTION

Ray tracing is a technique used to simulate the transport of light in digital images, and it involves computationally intensive processes. This technique can model various optical effects, such as reflection, refraction, and scattering. In the past, its applications were limited by the significant processing time required. However, in recent years, the introduction of hardware acceleration has enabled ray tracing to be used in real-time applications, including video games. In these contexts, optimizing ray tracing speed becomes critical, thereby opening the doors to harnessing parallelism more effectively.

3 PROBLEM STATEMENT

The first book of the ray tracing series introduces *path tracing*. The path-tracing process unfolds as follows:

(1) Calculate the ray from the eye to the pixel.
(2) Determine which object the ray intersects.
(3) Keep scattering until reaching maximum depths or not hitting.



(a) Sampling 10 times **(b) Sampling 500 times**

Figure 2: Noise and sampling

(4) Aggregate the color of all intersection points.

The steps above are repeated multiple times, known as *sampling*. It is crucial to note that path tracing employs a Monte Carlo method, resolving the rendering equation by finding a numerical solution to its integral. As the number of samples increases, the resulting ray approaches its actual value.

Ray tracing experiences a notable slowdown primarily due to the necessity of performing the process on all pixels. To illustrate, consider an image with dimensions 1,280 by 720, totaling 921,600 pixels — a substantial workload. Compounding this, the ray may scatter multiple times, reflecting its optical properties. Moreover, the challenge is further exacerbated by the need for numerous sampling iterations. To mitigate noise resulting from insufficient sampling, one often has to sample several hundred times. Figure 2a is the result of sampling ten times, which exhibits apparent blurs, while figure 2b below samples 500 times, leading to a much clearer result.

4 PROPOSED SOLUTION

Since the scene does not change throughout the ray tracing, each pixel can be rendered independently. This allows assigning pixels to multiple threads to achieve better performance. Algorithm 1 illustrates how ray tracing is done sequentially, representing the original implementation. The nested loop on lines 1 and 2 iterates over $W \times H$ pixels. Each pixel is sampled S times by the loop on line 4. After taking the average on line 10, the pixel is directly written to the file on line 11.

We propose two implementations: one parallelizes ray tracing using CPU multi-threading with OpenMP, and the other offloads the work to the GPU with CUDA. Both primarily focus on parallelizing the nested loop; other challenges and optimizations are discussed in Section 5.

4.1 OpenMP

Algorithm 2 shows the parallelized ray tracing with OpenMP. We add a *pragma* on line 3, indicating that the loop below is the target for parallelization. Additionally, the direct write has to be eliminated since the colors of pixels have to be written out in order, incurring a non-parallelizable dependency. We place them into a global array and write them out later after all pixels have been colored.

Algorithm 1: Ray Tracing on All Pixels — Serial

```
1  for j ← H to 0 do
2      for i ← 0 to W do
3          pixel ← black;
4          for s ← 0 to S do
               /* Ray goes randomly through the pixel.
                  */
5              u ← (i + RandomDouble())/(H − 1);
6              v ← (j + RandomDouble())/(W − 1);
7              ray ← GetRay(u, v);
               /* RayColor determines which object the
                  ray intersects and keeps scattering
                  until reaching maximum depths D or
                  not hitting.                        */
8              pixel ← pixel + RayColor(ray, WORLD, D);
9          end
10         pixel ← pixel/S;
11         WriteColor(pixel);
12     end
13 end
```

Notice that the random function is no longer the one in the serial algorithm 1; instead, `RandomDoubleR(seed)` is used. The former uses `rand()` internally, which has no guarantee on thread-safety[1]. We utilize `RandomDoubleR(seed)` with the thread-safe version `rand_r(seed)`. Each thread maintains its own `seed`, thus eliminating the potential for a race condition. The random seed is copied into the threads with `firstprivate(seed)`. Last but not least, scattering randomly implies imbalanced workload, and the density of objects varies across the scene. Different scheduling approaches can be applied to address the load balancing problem. We evaluate them in Section 7.1.

[1]Some implementations, such as *glibc*, have `rand()` being thread-safe by using locks. However, this can cause significant overhead when using multiple threads. Thus, the lock-free `rand_r(seed)` is still preferred.

Algorithm 2: Ray Tracing on All Pixels — OpenMP

```
seed ← some integer;
pixels ← array of size W × H;
#pragma omp parallel for schedule(static, 1) firstprivate(seed)
for j ← H to 0 do
    for i ← 0 to W do
        pixel ← black;
        for s ← 0 to S do
            u ← (i + RandomDoubleR(seed))/(H − 1);
            v ← (j + RandomDoubleR(seed))/(W − 1);
            ray ← GetRay(u, v);
            pixel ← pixel + RayColor(ray, WORLD, D);
        end
        pixel ← pixel/S;
        pixels[j × W + i] ← pixel;
    end
end
/* Write out the color of pixels.                */
for j ← H to 0 do
    for i ← 0 to W do
        WriteColor(pixels[j × W + i]);
    end
end
```

Algorithm 3: Render — CUDA

```
__global__ void Render(buffer, W, H, WORLD, camera, D, S,
  rand_states) is
    j ← threadIdx.y + blockDim.y ∗ blockIdx.y;
    i ← threadIdx.x + blockDim.x ∗ blockIdx.x;
    if i ≥ W ∨ j ≥ H then
        return;
    end
    idx ← j × W + i;
    pixel ← black;
    for s ← 0 to S do
        u ←
          (i + RandomDoubleC(rand_states[idx]))/(H − 1);
        v ←
          (j + RandomDoubleC(rand_states[idx]))/(W − 1);
        ray ← GetRayC(u, v, rand_states[idx]);
        pixel ← pixel +
          RayColor(ray, WORLD, D, rand_states[idx]);
    end
    pixel ← pixel/S;
    buffer[idx] ← pixel;
end
```

4.2 CUDA

To offload the work and execute them in parallel on the GPU using CUDA, first, we have to separate the code to be run on the GPU from the CPU code. The nested for loop is converted into a *global* function, `Render`, illustrating in algorithm 3. Each of the CUDA threads executes it.

For the memory allocations on the GPU, we analyze the places where each is used and placed in the corresponding positions: CPU, GPU, or shared. The pixel buffer is computed by the GPU and written out by the CPU. We allocate on Unified Memory with `cudaMallocManaged`; The random states are used only in kernel functions, thus allocating on the GPU with `cudaMalloc` suffice. The `WORLD`, i.e., the hittable objects, are somehow different. If we are to initialize it on the CPU, the objects must be allocated on unified memory so that both the CPU and GPU can share it. To do so, we have to allocate them with `cudaMallocManaged` instead of `new`, and also in the implementation of related classes, i.e., we cannot use `delete` in their destructors, which doesn't seem sound.

Notably, the operators `new` and `delete` can be used on both CPU and GPU, but they have different meanings. A `new` on the CPU has the memory allocated on the CPU, and the corresponding `delete` can only be called on the CPU, and vice versa. Knowing this, we also have another solution: initialize the world on the GPU. We then prepare another random state to use in the initialization of the world.

While adding the function execution space specifiers to have the function execute on the GPU, we encountered three types of functions that couldn't be directly converted due to their reliance on functions implemented by the standard library. These functions are the random generating function, `std::shared_ptr`, and `std::vector`.

4.2.1 Random Number Generation. The random function designed for the CPU cannot be used on the GPU. Fortunately, CUDA does support random number generation on the GPU through its API [2]. We utilize `curand_uniform_double(state)` as a replacement for `rand()` or `rand_r(seed)`, which requires a `curandState` to store the state. Each thread holds its own state. All functions that call the random generation functions are modified to take this additional parameter, such as the function `RandomDoubleC`, `GetRayC`, and `RayColorC`.

4.2.2 Smart Pointer. A smart pointer handles memory resources without the need for intervention by programmers. While it is not designed to be used on the GPU, CUDA doesn't currently have its own implementation. So, we have to use raw pointers and remember to release the memory resources explicitly.

4.2.3 Standard Container. Standard containers cannot be used in kernel function because their functions are not specified with `__device__`. We'll have to use C-style arrays, particularly for storing the objects.

Due to hardware limits, some patterns cannot be adopted to execute on the GPU.

4.2.4 Virtual Function. It's *Undefined Behavior* to pass an object of a class with virtual functions as an argument to a `__global__` function since one cannot dispatch a function from device to host.

[2] https://docs.nvidia.com/cuda/curand/device-api-overview.html#device-api-overview

To address this, classes are transformed to collaborate with others through *composition* instead of *inheritance*.

4.2.5 Recursive Function. The original serial implementation scatters the ray until it reaches the maximum depth or does not recursively hit any objects. This is due to the memory limit of the GPU. Although CUDA provides an API[3] to set the limit, it is likely to use up the memory since there are numerous threads. The recursive scattering function is rewritten into an iterative form to resolve the issue.

5 CHALLENGE

5.1 Loop for sampling N times

Since we utilize Monte Carlo methods for pixel rendering, this approach relies on sufficient randomness. Thus, the random sequence must be kept, indicating the existence of loop dependency. If we employ `firstprivate(seed)` in OpenMP to make all samples have the same seed, the pixel rendering may fail to converge, resulting in a noisy image with artifacts. Figure 3a is one example.

Later, we attempt to use different seeds for each sample. When a ray hits an object and needs to be reflected, it requires uniformly sampling points on the hemisphere centered around the hit point to generate the corresponding ray. Therefore, using different seeds for each sample causes the sampled distribution to be non-uniform, leading to one object having another shadow direction; Figure 3b is one example.

6 EXPERIMENTAL METHODOLOGY

Based on version 3 of the serial implementation[4], we implement the OpenMP and CUDA implementation and follow the following default settings.

- Image size: 1200×675
- Map size (Number of objects): $22 \times 22 = 484$
- Samples per pixel: 10
- Max depth: 50

For the hardware environment, we test on two NUMA nodes, Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz[4], 128GB RAM[5], and a single NVIDIA RTX 2080Ti 12GB. For the software environment, we set up a container on docker[6] with rootless mode as our evaluation platform. The operating system and Linux kernel are Ubuntu 22.04 and 5.15.0-87-generic. The versions of OpenMP and CUDA are 4.5 and 12.3, respectively. We compile the program with GCC 11.4.0.

6.1 Benchmark OpenMP with different threads

This test aims to analyze how much the performance can be improved with different threads. We plan to evaluate with four different settings:

- Collapsing pixel loops[7]
- Row-wise parallelization[8]
- Row-wise parallelization with cyclic partition
- Row-wise parallelization with cyclic partition and dynamic scheduling

For the measurement, we measure it with a speed-up metric over the serial implementation.

In addition, due to the communication overhead between two NUMA nodes, we have to limit the threads allocated to the same node. Thus, the maximum number of threads we can use is 20.

6.2 Benchmark CUDA with different block sizes

This test aims to analyze how much the performance can be improved with varying block sizes. For the measurement, we measure it with a speed-up metric over the serial implementation.

6.3 Benchmark with different map sizes

This test aims to analyze how much the performance can be improved with different map sizes or called number of objects.

We will select the best performance for the settings from the previous two tests.

For the measurement, we measure them in execution time.

7 EXPERIMENTAL RESULTS

7.1 Benchmark OpenMP with different threads

Figure 4a indicates row-wise parallelization with cyclic partition and dynamic scheduling in about 22 threads achieves the best performance. It also shows the existence of a load-balancing problem.

During 7 and 8 threads, the tendency of speed-up does not monotonically increase. We think the workload among threads is perfectly distributed. In other words, each thread proceeds to work at nearly the same time. With more threads, the speed-up still increases, which means the workload among threads overlaps. Thus, it proves that our thought is correct.

Because dynamic scheduling means distributing chunks to threads without any order and performs better load-balancing, it is reasonable that dynamic scheduling is better than static scheduling.

7.2 Benchmark CUDA with different block sizes

Figure 4b indicates the block size 8×8 performs the best. However, after investigating by the nvprof tool, we find that the maximum number of used registers per thread is 96.

According to the specification of RTX 2080Ti and 7.5 compute capability, 65,536 registers and 32 wraps are available per SM. Under the condition of using 96 registers per thread, only 20 wraps per SM are used. In other words, the maximum occupancy of SM is only 0.625. Thus, the block size 8×8 reaches the maximum occupancy.

7.3 Benchmark with different map sizes

Figure 4c shows the CUDA implementation performs the best. The serial implementation grows on a linear scale.

8 RELATED WORK

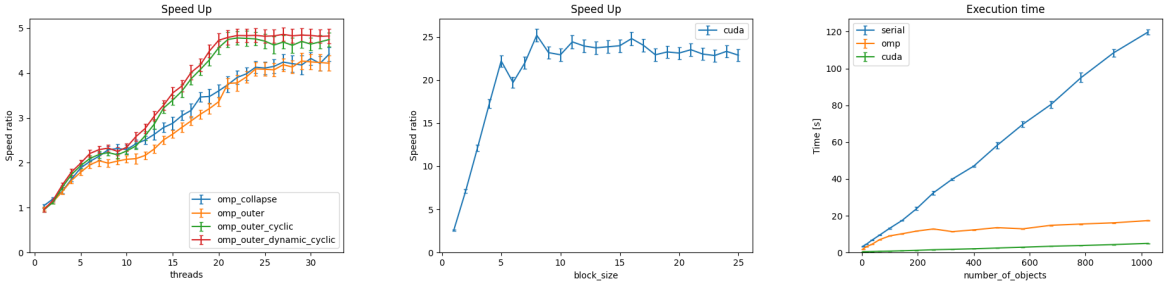
Several works have been done in OpenMP, CUDA, and OpenCL libraries. The RTFoundation[5] project following C++ standard is implemented in OpenMP and CUDA. The raytracing[2] project is implemented in OpenCL. The NVIDIA employee Roger Allen wrote a blog[1] about accelerating the program by using CUDA.

[3] Particularly, `cudaDeviceSetLimit`
[4] The CPU contains ten cores and twenty threads
[5] DDR4 2666MHz
[6] docker engine 24.0.6
[7] parallelize the width and height dimension
[8] parallelize the height dimension only

(a) Image with same seed sampling

(b) Image with different seed sampling

Figure 3: Different error sample methods



(a) Speed-Up with different threads

(b) Speed-Up with different block sizes

(c) Execution time with different map sizes

Figure 4: Benchmark results

9 CONCLUSIONS

In conclusion, we implement two parallelization approaches for ray tracing: OpenMP and CUDA. In the case of CUDA methods, the maximum speed up of CUDA over the serial program is 25x. While for OpenMP, the maximum speed up of OpenMP over the serial program is 5x. Besides, we tested different schedules in the OpenMP part. We found that appropriate load balancing can be 1.25 faster than the imbalance one.

The ray tracing method is based on Monte Carlo sampling. It relies on the proper random sequence. Thus, the sampling cannot be further parallelized.

Although the amplitude of improvements is not very large, it still has potential work that can be done in the future.

10 FUTURE WORK

We have investigated another hot spot in Table 1, *hit* function in Figure 5. Every pixel will call the *hit* function multiple times. We think it is worth potential work for acceleration.

However, it requires almost a square of CPU cores and nested parallelization because it is nested in the parallel region of rendering loops. Thus, we propose two possible approaches to solve this problem:

(1) OpenMP with NUMA nodes
(2) CUDA with Dynamic Parallelism

Find Closet Problem

```
bool hittable_list::hit(const ray& r, real_type t_min, real_type t_max,
                        hit_record& rec, unsigned int& seed) const
{
    hit_record temp_rec;
    auto hit_anything = false;
    auto closest_so_far = t_max;
    for (int i = 0; i < objects.size(); ++i)
    {
        const auto& object = objects[i];
        if (object->hit(r, t_min, closest_so_far, temp_rec, seed))
        {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
    }
    return hit_anything;
}
```

Figure 5: *Hit* function. find the closest intersected object.

REFERENCES

[1] Roger Allen. 2018. Accelerated Ray Tracing in One Weekend in CUDA. https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/
[2] Carter Roeser. 2019. raytracing. https://github.com/cdgco/raytracing.

Table 1: Time portion spent in each function (Top 5)

Function name	Time (%)	# Calls	Self / Call (μs)	Total / Call (μs)
`sphere::hit`	91.37	1924751405	0.04	0.04
`hittable_list::hit`	6.41	558455	9.73	148.35
`lambertian::scatter`	0.55	327140	1.42	1.42
`metal::scatter`	0.09	80146	0.94	0.94
`dielectric::scatter`	0.05	45906	0.87	0.87

[3] Peter Shirley. 2020. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html
[4] Peter Shirley. 2020. Ray Tracing in One Weekend (source code). https://github.com/RayTracing/raytracing.github.io/tree/release/v3.
[5] SeungWoo Yoo. 2021. RTFoundation. https://github.com/DveloperY0115/RTFoundation.