Parallelized Ray Tracing In One Weekend

組員:

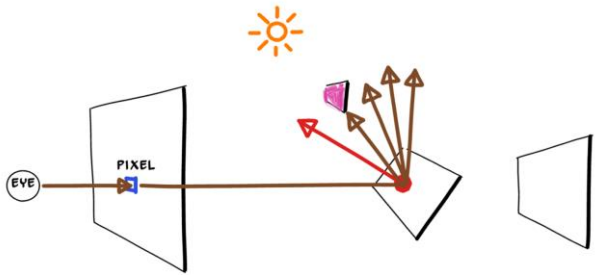
吳俊宏 109550165
應耀德 311553007
賴奕廷 312551073

Ray Tracing in One Weekend Book Series

https://raytracing.github.io/v3/books/RayTracingInOneWeekend.html

What Ray Tracing is Doing?

- Simulate the rays that comes into our eyes
- Model various optical effects: reflection, refraction, scattering



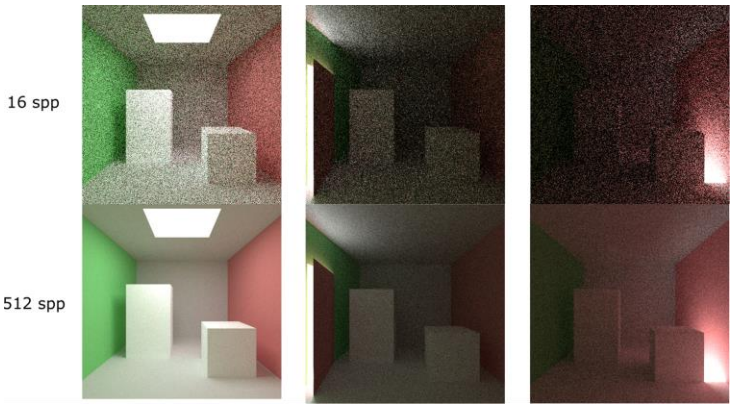
McGuire, Shirley, and Wyman, *Introduction to Real-Time Ray Tracing*, 28 Jul 2019, SIGGRAPH'19 Courses

How Ray Tracing Works?

Randomly sample N times from step 1 to 4 (Monte Carlo)

1. Calculate the ray from the eye to the pixel
2. Determine which object the ray intersects (closet one)
3. Keep scattering until max depths or not hitting (load balancing problem)
4. Aggregate the color of all intersection points

Average them

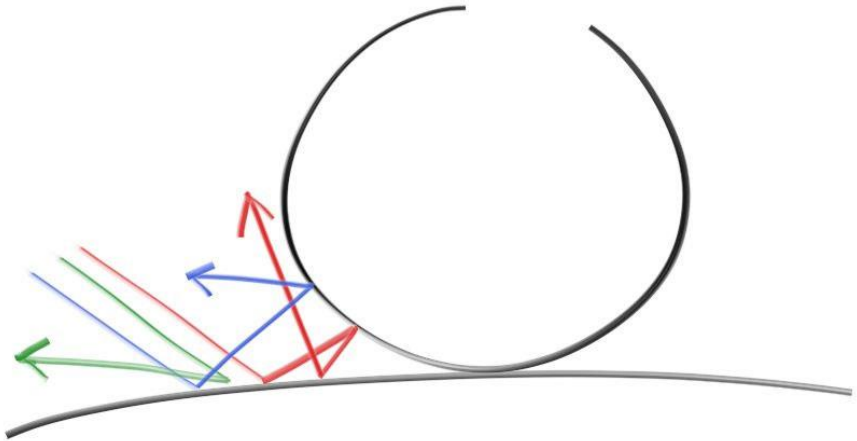


MONTE CARLO PATH TRACER:
https://maurocomi.com/monteCarloPathTracer.html

What's the Problem?

1. A ray may made multiple intersections since it scatters around
2. Computations on each pixels, e.g., 1,280 × 720 = 921,600

⇒ Very heavy workload!



Good News!

- Pixels are independent with each other

Ray Tracing on All Pixels

```
for (int j = image_height - 1; j >= 0; --j) {
    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);
    }
}
```

Cannot parallelize!

Proposed solution

- OpenMP
- CUDA

OpenMP

- Write each pixel to an array to eliminate the non parallelizable I/O.

```
#pragma omp parallel for schedule(static, 1) firstprivate(seed)
for (int j = 0; j < image_height; j++) {
    for (int i = 0; i < image_width; i++) {
```

CUDA

- CUDA device generates random number with curand_init. An independent random state is prepared for each thread.
- The scene has to be initialized on the device since it cannot be easily copied.
- It's *Undefined Behavior* to pass as an argument to a __global__ function an object of a class with virtual functions since one cannot dispatch a function from device to host.
- Recursive function causes *stack overflow*. The coloring function has to be converted into iterative form.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-functions

What else? We try...

- Parallelize loop for sampling N times

Loop for sampling N times

Ray Tracing on All Pixels

```
for (int j = image_height - 1; j >= 0; --j) {
    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);
    }
}
```

Loop for sampling N times

Seed Problem

```
1 #pragma omp parallel for reduction(pixel_color) firstprivate(seed)
2 for (int s = 0; s < samples_per_pixel; ++s)
3 {
4   auto u = (i + random_real_r(seed)) / (image_width - 1);
5   auto v = (j + random_real_r(seed)) / (image_height - 1);
6   ray r = cam.get_ray_r(u, v, seed);
7   pixel_color += ray_color(r, world, max_depth, seed);
8 }
```

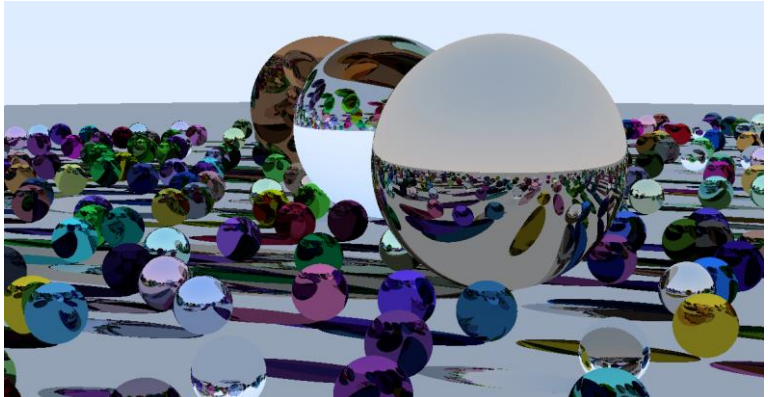
Loop for sampling N times

- *Monte Carlo* method requires enough randomness. The random sequence have to be kept.

  ⇒ Have loop dependency!

- While `firstprivate(seed)` makes all samples have the same seed,

  ⇒ Cannot converge!



Platform

Hardware	
CPU × 2 (*NUMA* node)	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz 10 cores 20 threads
GPU	NVIDIA RTX 2080Ti 12GB
RAM	128GB
Software	
OS	Ubuntu 22.04
Linux Kernel	5.15.0-87-generic
OpenMP	4.5
CUDA	12.3
Container	Rootless Docker

Settings

- Image size: 1200 x 675
- Map size(# of objects): 22 x 22 = 484
- Samples per pixel: 10
- Max depth: 50

OpenMP

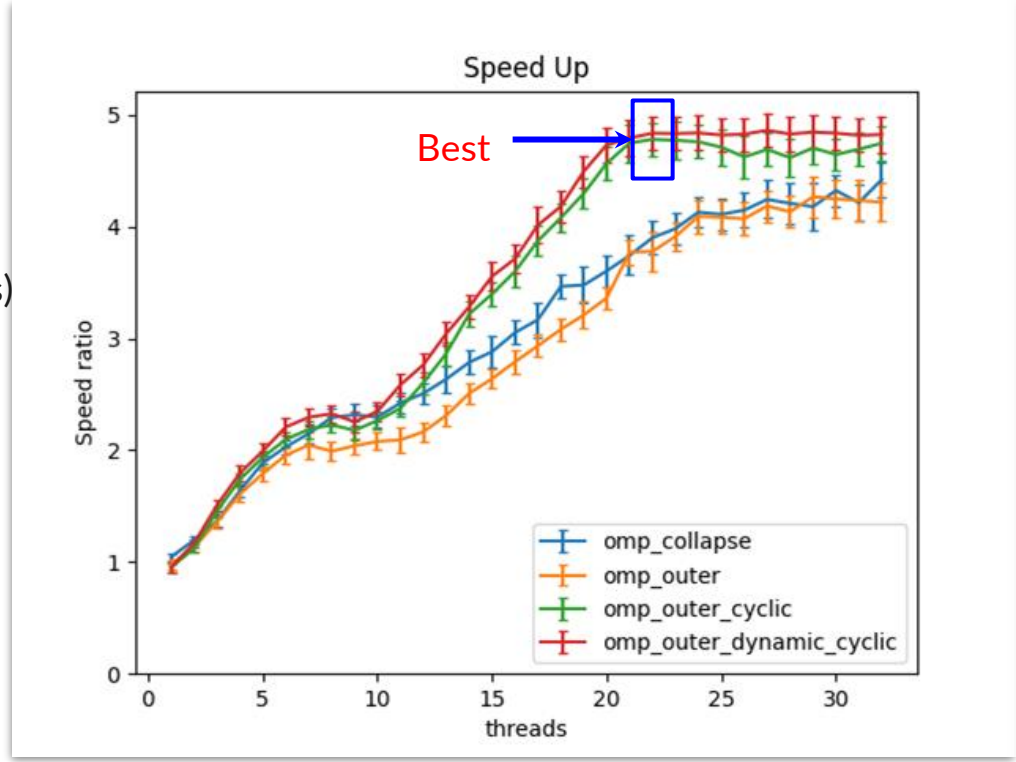
CPU limitation

- 2 NUMA nodes (10 cores 20 threads)
  ⇒ communication overhead

Program

- Load balancing problem

⇒ **Outer loop**
+ Dynamic scheduling
+ Cyclic partition is the best.



CUDA

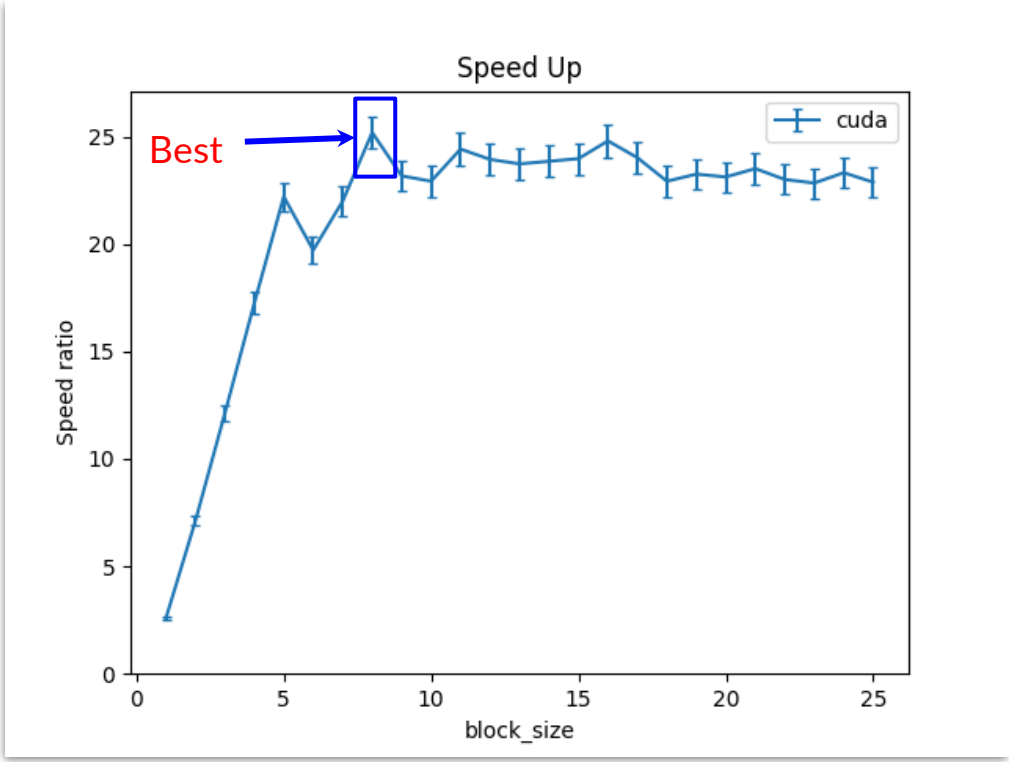
GPU limitation

- 65, 536 registers per SM
- 32 wraps per SM

Kernel function

- Use 96 registers per thread
- Can only use 20 wraps
- Max occupancy of SM is 0.625

⇒ **Block size 8 x 8 is the best.**



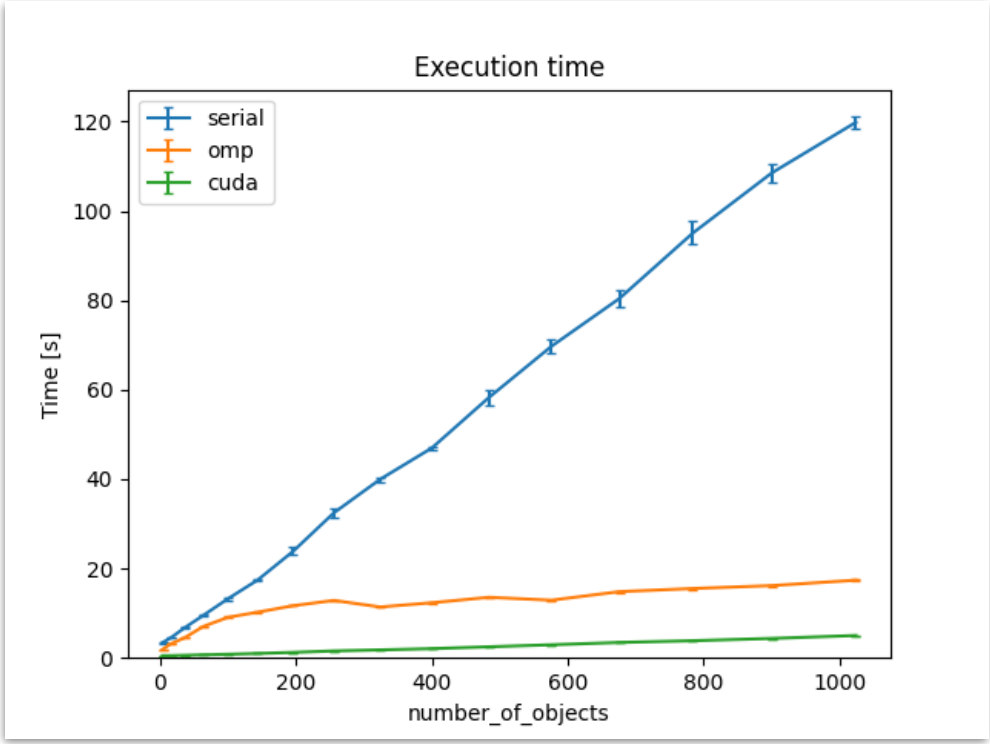
Map size (# of objects)

OpenMP

- Use 22 threads

CUDA

- Use 8 x 8 block size



Related work

- Accelerated Ray Tracing in One Weekend in CUDA | NVIDIA Technical Blog
- Ray Tracing in One Weekend Series
- eduardohenriquearnold/raytracer: Ray Tracer implementation from scratch based on Peter Shirley's 'Ray tracing in one weekend' (github.com)

Contributions of each member

- 吳俊宏 109550165: OpenMP Implementation 33%
- 應耀德 311553007: Support + Profiling + Benchmark 33%
- 賴奕廷 312551073: CUDA Implementation 33%

Conclusion

- The maximum speed up of OpenMP over the serial program is ~5x.
- The maximum speed up of CUDA over the serial program is ~25x.
- Appropriate load balancing can be 1.25x faster in comparison with the imbalance one.

- *Monte Carlo* method relies on the proper random sequence, thus the sampling cannot be further parallelized.

Future work

- Parallelizing finding the closest object
  - Need nested parallelization
  - Require more CPU cores

  ⇒ OpenMP with NUMA nodes
  ⇒ CUDA with Dynamic Parallelism

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

Find Closet Problem

```
1  bool hittable_list::hit(const ray& r, real_type t_min, real_type t_max,
2                          hit_record& rec, unsigned int& seed) const
3  {
4      hit_record temp_rec;
5      auto hit_anything = false;
6      auto closest_so_far = t_max;
7      for (int i = 0; i < objects.size(); ++i)
8      {
9          const auto& object = objects[i];
10         if (object->hit(r, t_min, closest_so_far, temp_rec, seed))
11         {
12             hit_anything = true;
13             closest_so_far = temp_rec.t;
14             rec = temp_rec;
15         }
16     }
17     return hit_anything;
18 }
```

End