Parallel Ray Tracing with OpenMP

Hanlin He, Wenbo Song, Yaowei Zong

Abstract

Ray tracing has been a well-known effective approach for rendering an image. However, the high computation complexity of this approach has been an obstacle to several real world applications in the industry. With the development of computing power and processor architecture in recent years, it becomes possible to distribute this task over multiple computing units to accelerate the algorithm. In this project, we implemented a SIMD parallel ray tracer running on multi-core CPUs with OpenMP. And we managed to optimize this parallel ray tracer by designing a load balance method and performing a cache optimization. In the end, our optimized algorithm achieved a significant speedup in comparison to the sequence version and demonstrated a similar performance with the CUDA version.

1 Introduction

Ray tracing is one of the major approaches for realistic rendering of images in the field of computer graphics. It is a simple yet powerful technique due to its ability to simulate different physical effects with high quality such as to render shadows, reflections, and refracted light [1].

Ray tracing has been widely used in industries like animation. However, its expensive computation cost appears to be seriously handicapped when it comes to many other applications especially the more interactive ones like gaming. Therefore, the calculation speed of the ray-tracing method is undoubtedly one of the basic problems that must be dealt with.[2]

A basic ray tracing algorithm always iterates through every pixel of the image to calculate its final color[3]. In each iteration, first, the viewing ray is calculated. Then compute the first object hit by the ray and its surface normal. With reflection or refraction, it may require to generate a new ray from the hit point. Finally, set the color of the current pixel to the value computed from the hitting point, light, the object surface, and the contribution of reflecting/refracting rays.

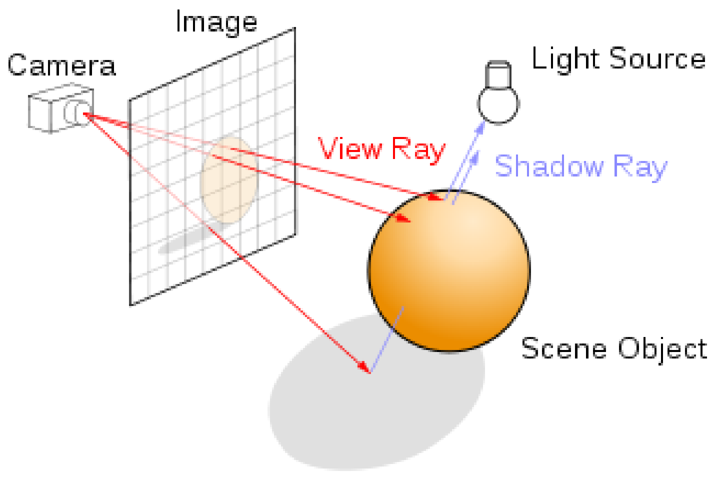

Figure 1: Ray Tracing

In this project, we tried to accelerate a ray-tracing program by parallelizing the pixel traversal using OpenMP to distribute the work among multiple threads.

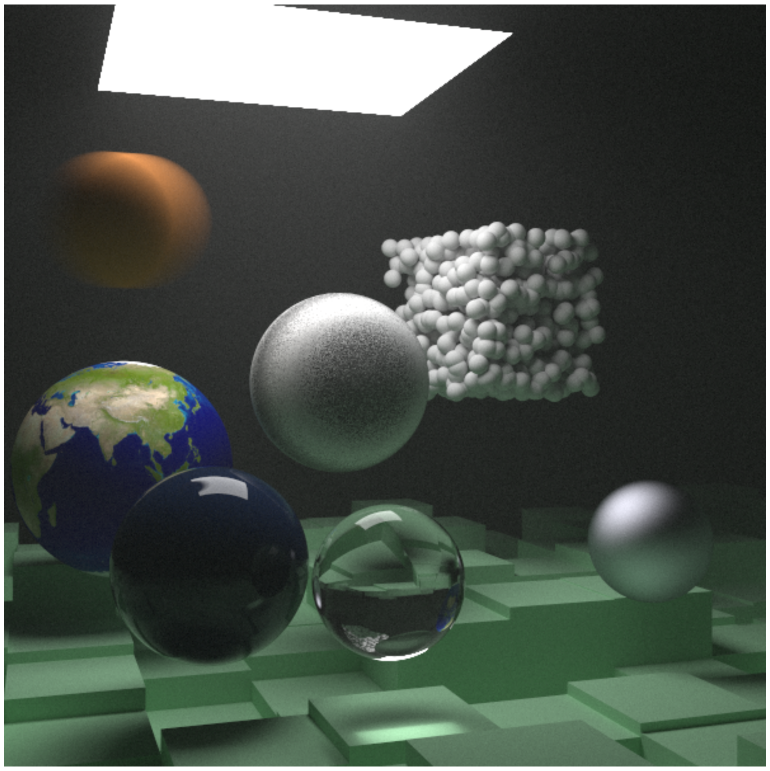

Figure 2: One Sample Image Produced by Our Program

2 Literature Survey

It has long been realized that parallel computers can be an effective way to accelerate the massive float point calculations of ray tracing[4]. Depending on which part of this problem is divided among processors, former studies can be categorized into two major areas: data-driven and demand-driven.

The data-driven approach describes the ones that distribute the object space over the processors. Rays are traced through each cells of the spatial subdivision which are assigned to different processors. This approach was firstly proposed by Swensen et al[5] and further refined by Cleary et al[6]. Despite this approach can accomplish some speedup, it inherently has several severe disadvantages. Kobayashi et al[7] and Jenson et al[8] proposed some load balancing strategies to mitigate the unbalance load problem caused by the clustering of rays and their reflections. However, apart from the load balancing problem, there are more problems including the extra communication overhead, which will become a bottleneck to further accelerate ray tracing.

Another popular approach is to assign each processor with the task to calculate different parts of the final image. This is called the demand-driven approach[9] and it is the basis of our parallel implementation. Crow et al[10] implemented several demand-driven approaches using different image splitting logistics in 3-D object space. Later, another efficient implementation was proposed[11], which distributed the pixels of the image to SIMD processor arrays. In spite of the fact that this approach can relatively distribute the computation task more evenly among the processors than the former approach, the recursion depth when computing the color of each pixel can be largely different from each other. However, this problem has been neglected in the former studies, which would be our main focus in this project. Moreover, coherent subsections of the image might be assigned to different processors, which can result in cache misses.

3 Proposed Idea

3.1 Sequential version

The basic ray tracing algorithm continuously computes the color for each pixel. These pixels are independent of each other, so it is suitable for parallel.

```
#sequential part
#parallel part
for each pixel do
    compute viewing ray
    color = color_evaluate(ray, depth)
#sequential part
```

To simulate the effects of reflection and refraction, when a ray hits the object, it will shoot a new ray aimed in a different direction. The color of the computed pixel is the result of such a recursive ray tracing with a limited recursion depth.

```
color_evaluate(ray, depth)
    if (ray hits an object with t in [0, infinity)) then
        compute color
        if the depth > 0 and can produce new ray
            create new reflected/refracted ray
            color += color_evaluate(new_ray, depth-1)
    else
        color = background color
    return color
```

3.2 Basic Parallel Idea

We were implementing a SIMD parallel ray tracing running on multi-core CPUs with OpenMP. The basic idea is to assign the *for each* loop to different threads since these pixels are independent. In this way, the original sequential render program can be run in parallel.

```
#sequential part
#parallel part
#pragma omp parallel for
for each pixel do
    compute viewing ray
    color = color_evaluate(ray, depth)
#sequential part
```

We continued to optimize the running time for this parallel version in the following *Load Balance* and *Cache Line* sections.

3.3 Load Balance

The basic version is simple and can be optimized. The first issue we faced with is load imbalance. Obviously, there is a workload difference between different pixels due to the variance of needed recursion times. If a ray does not hit any object, then it will directly return the background color. Or if the material it hits does not have any reflection/refraction effect, then it will directly return the material's color. Since it is impossible to directly notice the imbalance problem from the *parallel for* part, we have to find out a criterion evaluating and balancing the workload. Otherwise, the work is distributed unevenly, which means some threads may finish earlier and then wait in idle, causing a performance loss.

In this project, we took two strategies to balance the load of different threads. The first one was creating an operation pool. Since we could not know the exact number of recursive computations of each pixel computation. We set up the operation pool by first adding all the rays emitted from the pixel points into it. Then we assigned these operations in the pool to several threads. After the first round computation finished, we got the newly produced rays and refilled them into the pool, repeating the previous steps (Figure.3).

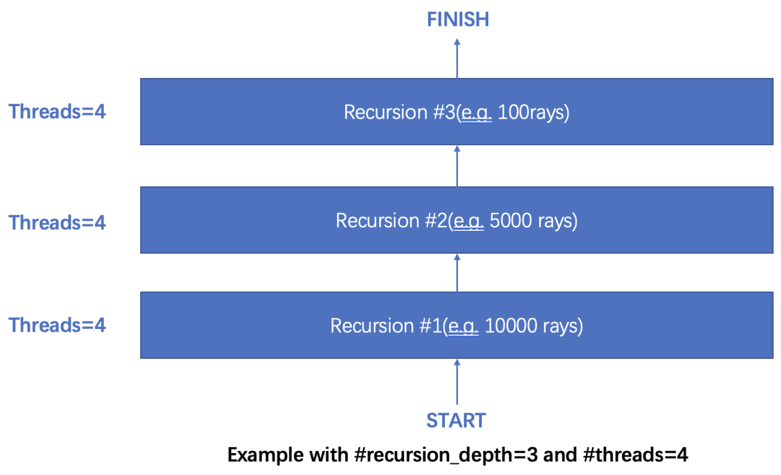
FINISH
Threads=4
Recursion #3(e.g. 100rays)
Threads=4
Recursion #2(e.g. 5000 rays)
Threads=4
Recursion #1(e.g. 10000 rays)
START
Example with #recursion_depth=3 and #threads=4

Figure 3: Load Balance with Operation Pool

Then the pseudo code changed as the following. We could also make use of multiple threads in creating operations into the pool. Compared with the basic version, this kind of load balancing could make full use of all threads, but it also created some overheads while creating and maintaining the operations in the pool.

```
#sequential part
#parallel part
for each pixel do
    compute viewing ray
    collect these rays and corresponding computation into the pool
#pragma omp parallel
while pool is not empty
    #pragma omp for
    for each operation in the pool
        pop the operation
        Evaluate the color and add it into the pixel matrix
        if produce new ray
            add new ray to the pool
#sequential part
```

Is there any better and simpler way to do load balance? Yes, that is *omp schedule*. In fact, we were not assigning the work to threads pixel by pixel, but line by line. The *Default* schedule would cut the whole image into sections and then assign them to different threads. What if we manually assign the image matrix line by line using *static,1*? The reason we selected it is based on the assumption that the neighbor pixels are more likely to have the same recursion depths. So when we distributed the work line by line, threads were more likely to get similar loads and thus reduce the performance loss caused by imbalance (Figure.4).

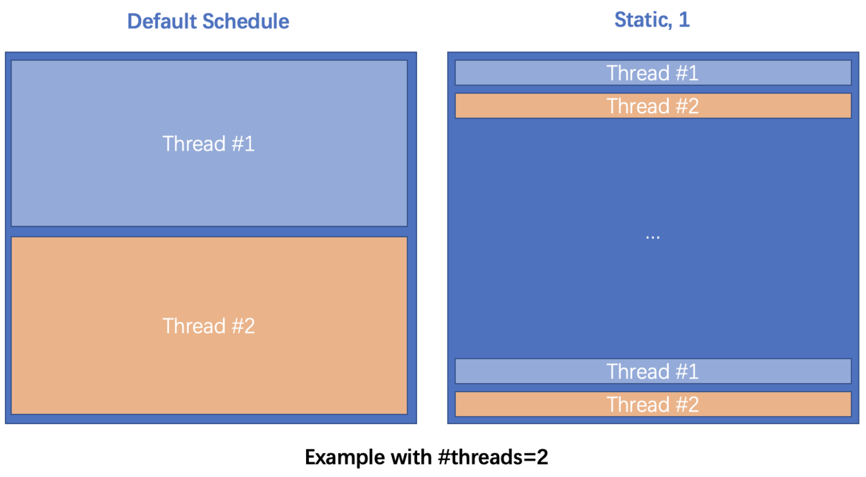

Figure 4: Load Balance with OMP Schedule

To justify our assumption, we compared the overall loads of each thread with different scheduling strategies (Figure.5). For example, with fixed *image_width=800*, and *#thread=8*, the load distribution varied a lot with the *default* schedule. With *static* schedule and chunk-size equals to one, the load distribution among threads was more balanced. The best distribution pattern happened while distributing the work to threads pixel by pixel. It could be implemented by transferring the original render into pixel by pixel version with the help of omp collapse. However, the line-by-line distribution was good enough. So in the experiment, we selected the line-by-line distribution with (static, 1) schedule.

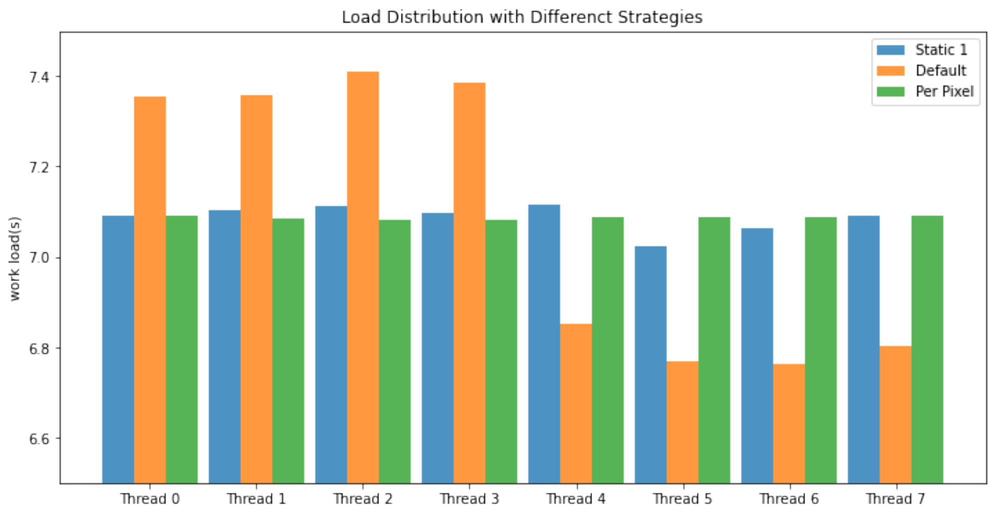

Figure 5: Load Distribution with Different Schedule Strategies

```
#sequential part
#parallel part
#pragma omp parallel for schedule(static,1)
for each line in the image do
    for each pixel in the line do
        compute viewing ray
        color = color_evaluate(ray, depth)
#sequential part
```

3.4 Cache Optimization

As mentioned before, we distributed the image by lines, which could be a potential bottleneck in running time. Because the image size varied, there might be cache misses when switching between threads. Moreover, three matrices (representing R, G, and B) were updated in each iteration for calculating the color of each pixel, which could also increase the caching problem since each thread had to fetch three different matrices.

Therefore, we modified the updating pattern for the color calculation, respectively the R, G, B matrices standing for every primary color. Instead of constantly fetching three matrices in each iteration, we decided to use a large matrix to store all the numbers and reconstruct it to put the R, G, B values of a fixed pixel nearby. In this way, with a single cache, we can update three values together.

Ideally, these changes should result in a speedup. Because given that each thread only needed to update one matrix, they could update all the three variables fetched from the memory. Although we would have to rebuild the three matrices from the large matrix after calculating the color of each pixel, we could parallelize this part to mitigate the time cost.

4 Experimental Setup

The scene we use for the following experiments consists of still and moving spheres on a checkered ground as shown in figure 6. Shading effect includes **diffusion, shadow, reflection, refraction, texture material, and motion blur**.

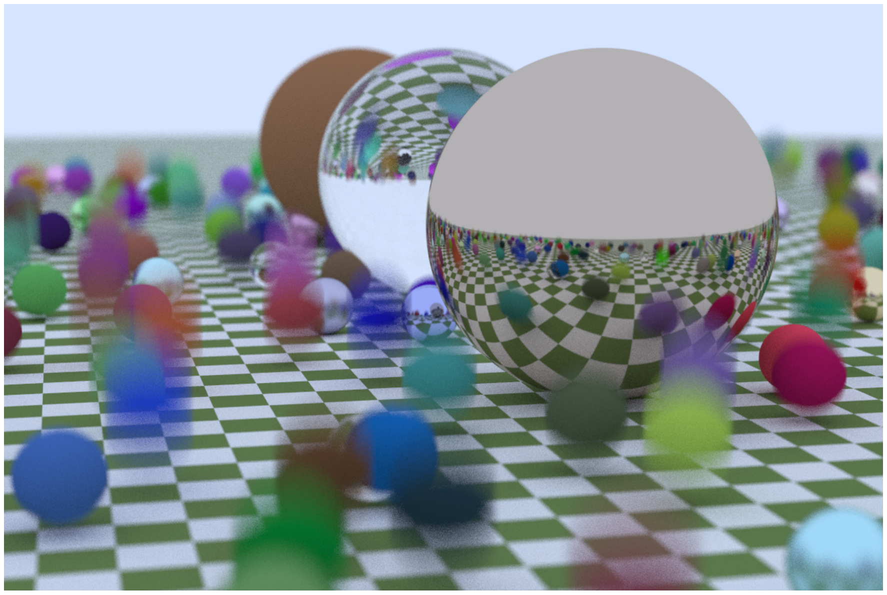

Figure 6: The Test Scene

The variables of this program includes **image_width**, **samples_per_pixel** (for fuzzy reflection and defocus blur), **max_depth** (for reflection and refraction) and **number of threads**. The combination of **image_width**, **samples_per_pixel**, **max_depth** decides the size of problem. With fixed **max_depth**, the problem size is linear with **image_width**^{2} $\times$ **samples_per_pixel**.

To test the scalability and speedup over sequential version, we run tests on NYU CIMS Compute Server *crunchy1.cims.nyu.edu*. This server has four AMD Opteron 6272 CPUs (2.1 GHz, 4*16 cores/threads, 64 cores/threads total). The compiler we use is GCC-11.2.0 by *module load gcc-11.2* on crunchy1. To use different versions of implementation, simply checkout the corresponding git branch and create a build folder, then use CMake-3 to compile the provided *CMakeLists.txt* file. The CUDA version was tested on NYU CIMS Compute Server *cuda1.cims.nyu.edu*.

The tools we used for performance analysis include recording the execution time using *omp_get_wtime()*, and checking cache-misses using the Performance analysis tools *perf*.

5 Experiments & Analysis

5.1 Sequential Version and Basic Parallel

First, we tested the basic parallel idea implemented in the master branch with *image_width=1200*, *samples_per_pixel=100*, *max_depth=5*. We use *num_threads=1* as the sequential version baseline. The speedup was calculated by *time using one thread / time using t threads*. Results are shown in Figure.7 and Table.1.

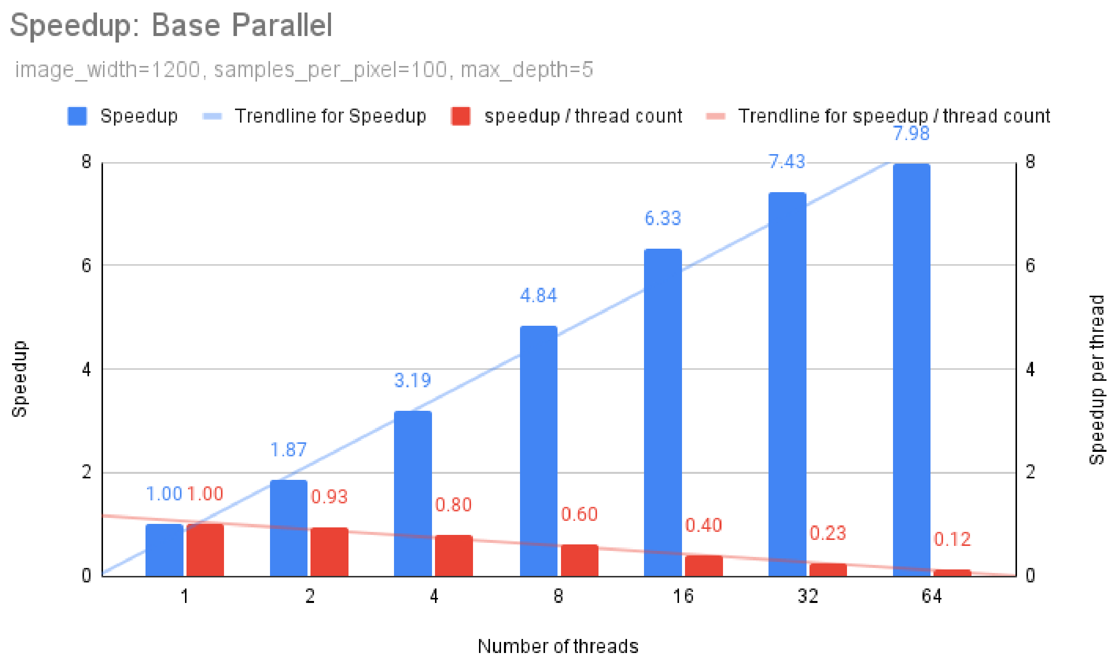

Figure 7: Speedup for Basic Parallel

We got a great speed-up, even in the basic version. The speed-up was not increasing linearly with the number of threads due to other overhead like creating and managing threads. However, it got a great result especially when the number of threads was small. For example, we got **1.869** speed-up with 2 threads, and **3.192** speed-up with 4 threads. When the number of threads exceeded 8, the accelerating effect won't significantly increase, with a *speed-up/thread* less than 0.5.

threads	real time (seconds)	speedup	efficiency
64	180.741	7.985	0.125
32	194.357	7.425	0.232
16	227.899	6.333	0.396
8	298.451	4.836	0.604
4	452.150	3.192	0.798
2	772.112	1.869	0.935
1	1443.194	1	1

Table 1: Speedup: Basic Parallel

5.2 Load Balance with Operation Pool

We tested the code maintaining a pool to distribute work among threads with fixed problem size and different numbers of threads and compared the results with the basic parallel version (Figure 8). The results shows, although we did accomplish acceleration in the parallel part with more threads, we also added some overheads while initializing and reconstructing the operation pool. The overall cost of using the operation pool was greater than the basic version in the master branch.

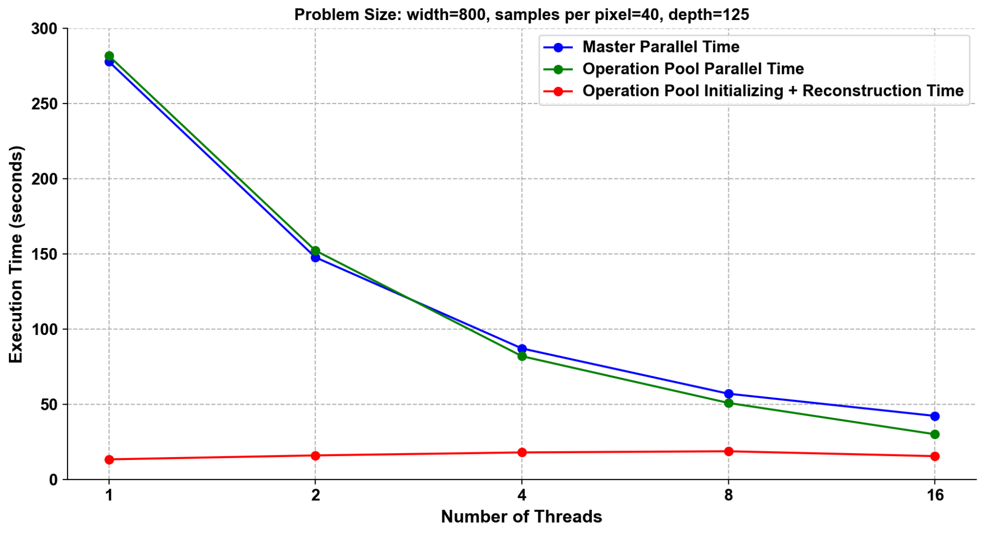

Figure 8: Speedup for Basic Parallel

Although this is a great idea to evenly distribute workload between threads, it requires other methods to reduce the overhead mentioned above. To be specific, if we could develop a better way to simulate the reflection and refraction without using recursion, then there is no need to reconstruct the pool. Instead, we can fetch the whole operations at the beginning, add them into the pool, and let threads pick tasks from it. That is so-called "Recursion Removal".

5.3 Load Balance with Schedule

As justified before, the nearby pixels are more likely to have same the recursion times so as the computation needed. In this section, we tested the parallel version with schedule *(static,1* to assign the work to threads line by line. Comparing Figure.9 with Figure.7, we can conclude that, the speedup with more threads is more significant with a balanced load distribution.

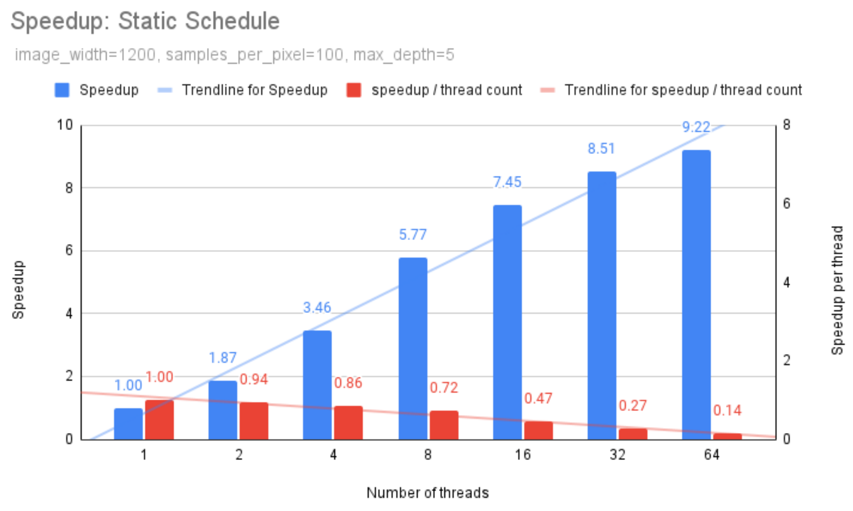

Figure 9: Speedup for Static Schedule

In Figure.10, we compared the time cost of parallel part of different versions. The basic version, the static schedule version, and the version of cache optimization with reorganized matrices. Among them, the static schedule one had the best time performance. In this scenario, the static schedule version could reach 15% speed-up in contrast to the basic version.

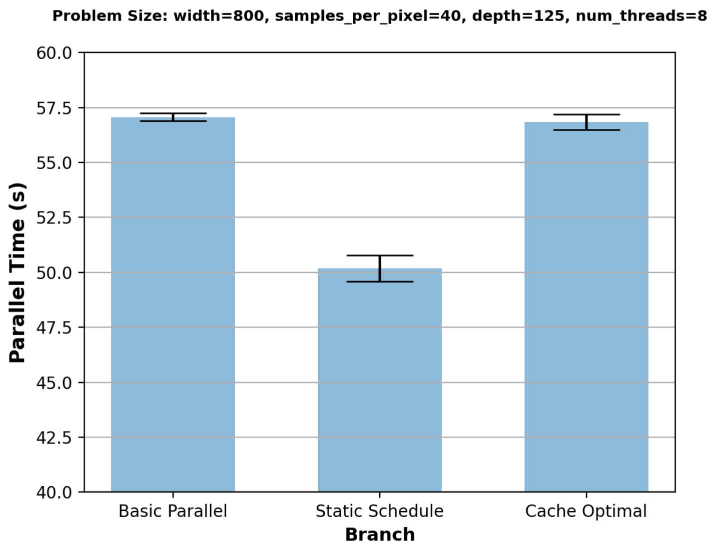

Figure 10: Parallel Time between different branches

5.4 Cache Optimization

The *perf* command on Linux helps us to monitor the performance statistics of the program, including cache-miss. In this section, we tested the code version in which a reorganized matrix could help to reduce the data that needs to be fetched in every iteration. As shown in Figure.11, the cache-optimized version indeed achieved a cache-miss-reducing from 11% to 8% compared with the basic version. However, this kind of cache optimization won't contribute much to the overall time cost according to our experiments in Figure.10.

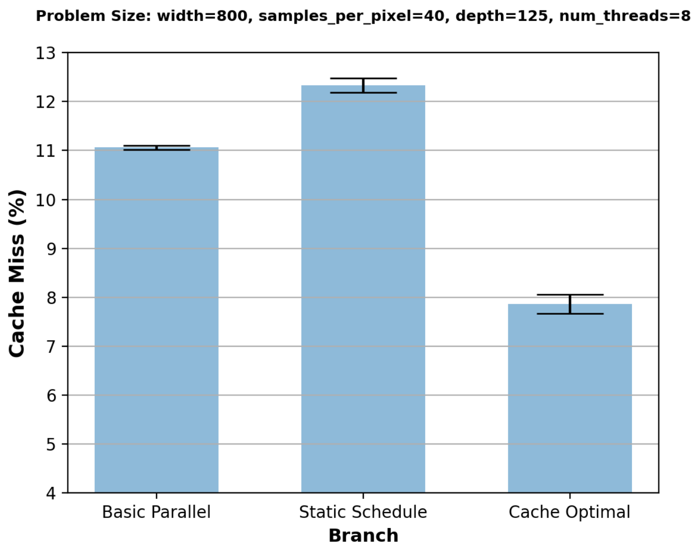

Figure 11: Cache-Miss Percentage between different branches

5.5 Scalability

At the end of experiments, we explored the scalability of our parallel ray tracing algorithm (Figure.12). We fixed the **max_depth** and the **samples_per_pixel**, then the problem size is in linear relationship with **image_width**^{2}. So if we set **image_width** as 400, 560, 800, 1200, and 1600, then the problem size is increasing exponentially.

The experiments results are shown in Figure.12. As the problem size increases, with the number of threads fixed, we could maintain a great speed-up. And as the number of threads increases, with the problem size fixed, we could achieve a higher speed-up. Besides, with a bigger problem size, the speed-up efficiency (speedup relative to 1 thread / #threads) increased too (Table.2).

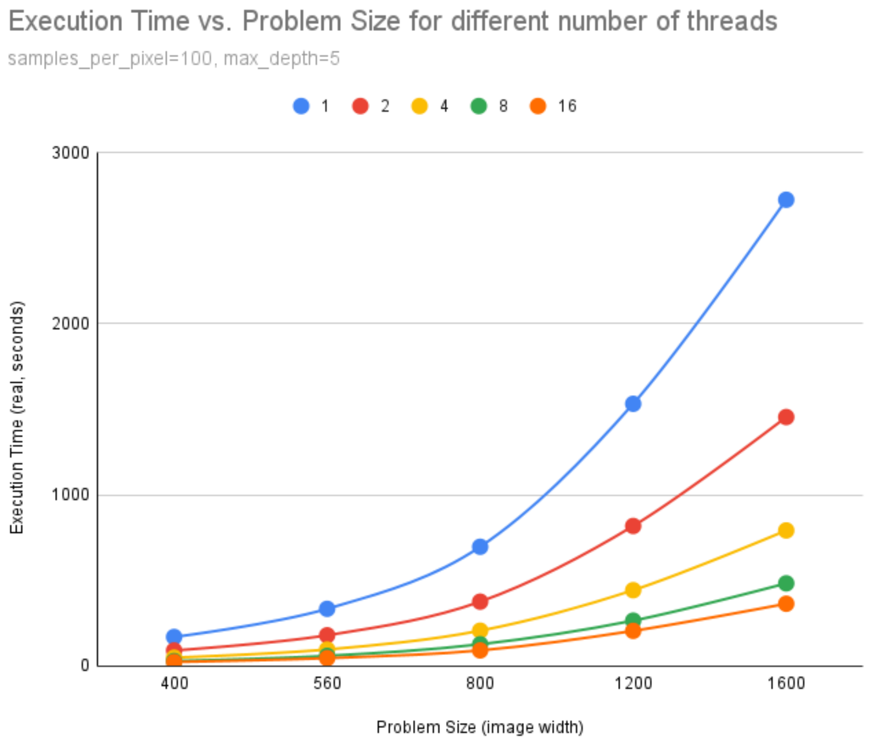

Figure 12: Execution Time vs. Problem Size

threads	400	560	800	1200	1600
1	1.0	1.0	1.0	1.0	1.0
2	0.932	0.929	0.926	0.936	0.936
4	0.857	0.862	0.842	0.864	0.860
8	0.717	0.698	0.684	0.721	0.704
16	0.442	0.450	0.475	0.465	0.467

Table 2: Efficiency

To compare our parallel ray tracing with other ones' implementations, we referred to a CUDA version and transferred it to our experimental environment. The execution time comparison is listed in Table.3.

Version	400	560	800	1200	1600
CPU 16 threads	23.91	46.20	91.22	204.90	362.88
GPU CUDA	12.13	19.20	36.19	73.94	127.73

Table 3: Image Rendering Time: CPU with 16 threads vs. GPU

6 Conclusions

In this project, we implemented a parallel ray tracing algorithm for multi-core processors by OpenMP. To optimize the program and boost acceleration, we experimented with the effects of load balance and cache-miss, and implemented different strategies. In the end, we tested the selected version with different problem sizes and #threads, compared with the results of the CUDA version.

- The ray tracing algorithm is a widely used computer graphics technology. The independence of pixel color computation makes it a perfect target for parallelizing. Many researchers have already implemented the parallel versions for GPU, however, parallel ray tracing for multi-core CPU has not been fully discussed. Parallelizing ray tracing for CPU is also meaningful because, in this way, some lightweight applications using ray tracing (e.g. graphic interface and human-computer interaction) can also benefit from the progress of multi-core processors nowadays.
- The load balance is the primary optimization source. There are some different strategies. First one is creating an operation pool and let threads pick tasks from it, but this required "Recursion Removal" otherwise there are some overheads while maintaining the pool. The simpler one which is suitable for OpenMP is making good use of the **omp schedule**. As justified before, nearby pixels (or lines) are more likely to have similar computation loads. In this way, a static schedule is good enough and won't add many overheads. The cache-miss can also be reduced if we took some strategies. However, in this problem, the performance loss caused by cache-miss is not significant, so is the speedup from this optimization.
- The parallel ray tracing implemented in this project has a great speed-up and efficiency, even in the comparison with the CUDA version. It is scalable with increasing problem size or #threads. Since the ratio of sequential is relatively low in ray tracing problem, we don't need too big a problem size to get a satisfactory speedup. It keeps providing fantastic speedup while increasing the problem size. In the meanwhile, it won't lose much efficiency while increasing the number of threads.

References

[1] GH Spencer and MVRK Murty. "General ray-tracing procedure". In: *JOSA* 52.6 (1962), pp. 672–678.

[2] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. "Arts: Accelerated ray-tracing system". In: *IEEE Computer Graphics and Applications* 6.4 (1986), pp. 16–26.

[3] Andrew S Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[4] Alan Heirich and James Arvo. "A competitive analysis of load balancing strategies for parallel ray tracing". In: *The Journal of Supercomputing* 12.1 (1998), pp. 57–68.

[5] Mark Dippe and John Swensen. "An adaptive subdivision algorithm and parallel architecture for realistic image synthesis". In: *ACM SIGGRAPH Computer Graphics* 18.3 (1984), pp. 149–158.

[6] John G Cleary et al. "Multiprocessor ray tracing". In: *Computer Graphics Forum*. Vol. 5. 1. Wiley Online Library. 1986, pp. 3–12.

[7] Hiroaki Kobayashi et al. "Load balancing strategies for a parallel ray-tracing system based on constant subdivision". In: *The visual computer* 4.4 (1988), pp. 197–209.

[8] Erik Reinhard and Frederik W Jansen. "Rendering large scenes using parallel ray tracing". In: *Parallel Computing* 23.7 (1997), pp. 873–885.

[9] David Plunkett and Michael Bailey. "The vectorization of a ray-tracing algorithm for improved execution speed". In: *IEEE Computer Graphics and Applications* 5.08 (1985), pp. 52–60.

[10] Franklin C Crow et al. “3d image synthesis on the connection machine”. In: *International Journal of High Speed Computing* 1.02 (1989), pp. 329–347.

[11] Tony TY Lin and Mel Slater. “Stochastic ray tracing using SIMD processor arrays”. In: *The Visual Computer* 7.4 (1991), pp. 187–199.