

Software Systems Assignment 2: Performance

Jacob Grønseth Olsson (6545270)

Soukarya Ghosh (5447593)

1 Introduction

In this assignment, we worked on improving the performance of a Rust raytracer. The original program was intentionally slowed down, and the goal of the assignment was not to optimize the raytracer, but to identify key performance problems, know how to fix them, and measure the effects of each fix.

We focused on five different kinds of optimizations to make the raytracer performance better. First, we removed unnecessary print logging in the most performance-critical parts of the code, because printing to the terminal for every row added a lot of I/O overhead. Another major change was switching from expensive system-based random number generation to a faster per-thread pseudo-random generator, which removed many costly system calls. Moreover, we also reduced lock contention in the thread generator, by locking the output buffer only once per row instead of once per pixel. Then, we improved memory usage by avoiding unnecessary cloning (ray) and other allocations (ray) in tight loops. Finally, we applied improvements like inlining frequently used vector math functions as well,

2 Experimental setup

The raytracing is ran and tested on a 16 core 13th Gen Intel® Core™ i7-1360P with 16GB RAM running Fedora Linux 42 (Workstation Edition). All benchmarks were run using Criterion in release mode. Profiling was done using `cargo flamegraph`, with the resolution reduced to 50×50 so that profiling runs remain short. This configuration was used for every benchmark.

3 Baseline

3.1 Description

The baseline is really slow, and the main thing that stands out is all the printing to the kernel. Terminal output is extremely slow compared to computation, and every print forces the thread to perform a blocking I/O operation. Since this happens for many rows and across several threads, a huge amount of time is wasted waiting for the terminal output.

3.2 Profiling

Since the baseline is very slow, we reduced the size of the ray tracer to be 50×50 . When profiling, it is clear that most of the time is spent writing, and syscall. This is the main bottleneck of the baseline, as you can see in figure 5. The other parts that the program spends time in are intersects and Arc. This is difficult to read from the flamegraph, but is seen when running "perf report". The report shows that the writing takes up

about 60% of the time, while the syscall at the end takes about 15% of the time.

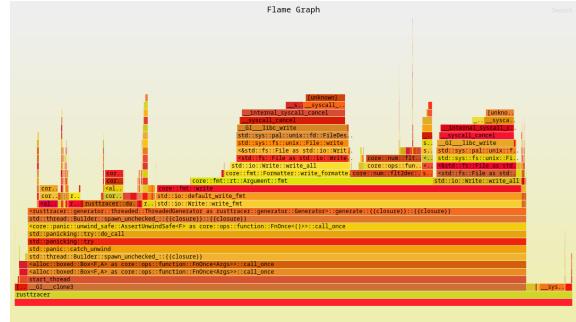


Figure 1: Initial profile of the ray traces, rendering a 50×50 image.

Overall, the raytracer takes about 73 seconds to render a 50×50 image in the baseline configuration.

4 Optimization I: Buffering/Removing I/O

4.1 Description

This optimization is directly based on the "I/O buffering" and "calling expensive operations less often" techniques from the lecture. Since the initial profile showed I/O to be the biggest bottleneck, we started by removing I/O. That means writing/reading from files or from the kernel. By removing thousands of terminal writes and file writes, the program spends almost all its time doing actual raytracing work instead of waiting on the operating system. The profiling results showed that the program printed a line for almost every pixel row, and in `outputbuffer.rs`, every call to `set_at` wrote the pixel to a file immediately. This means that the CPU spends most of its time waiting for the kernel instead of rendering rays. These are all not a part of rendering the image, so we can just remove the entire thing. If it was needed, we would instead like to buffer it, and write once at the end of all the calculations. There is also printing for every pixel in the file `mstracer.rs`. By removing this we clean up the kernel and remove unnecessary I/O.

4.2 Estimation

Based on the fact that the writing is about 60% and the syscall about 15%, removing the writing and printing, we estimate that removing this will reduce the time by 75%.

4.3 Profile/ Benchmark

Benchmarking shows that for rendering the 50×50 picture, removing I/O operations reduces the time from

73.77 seconds to 1.69 seconds. Which corresponds to a 97.7% improvement. Shown in table 5 and figure ??.

Total execution time in seconds	
Baseline	73.77
Optimization	1.69

Table 1: The performance of the baseline ray tracer compared with the optimized ray traces after I/O removal.

This improvement is better than what we expected. It seems like a 97.7 percent improvement. Even though during profiling it seemed like it was only 75% of the time. The reason is that the flamegraph only shows time spent in user-visible I/O operations, while there are additional operations that are not user-visible that also contribute to the slowdown.

Profiling now looks like figure 2

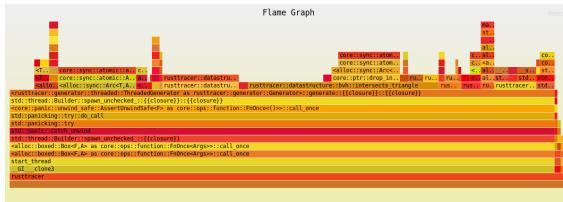


Figure 2: Profiling after I/O removal.

5 Optimization II

5.1 Description

This optimization is based on our own ideas and creative thinking. In the original ray tracer, every ray bounce and every Monte Carlo sample used OsRng or another OS-backed random number generator. OsRng relies on system calls to gather entropy from the operating system [1]. System calls are extremely slow compared to normal function calls, and the ray tracer calls the RNG thousands of times per frame. While profiling, we discovered that the original RNG was slow, so changing the RNG with a faster one would benefit the program a lot. So instead of using OsRng, we used SmallRng, a fast pseudo-random generator provided by the rand crate [1]. The OsRng is slow due to OS calls, but more accurate, while the SmallRng is only pseudo-random, but it is faster.

5.2 Estimation

No real estimation, since we dont know the performance difference of the two Rng's, but removing OS calls eliminates thousands of extremely slow kernel transitions. Hence, we think the fix can improve performance by quite a bit.

5.3 Profile/ Benchmark

The ray tracer finished in just 1.16 seconds, compared to the 1.69s after the previous optimization of removing I/O. Which is about a 31% improvement.

The profiling in this case now looks like shown in figure 3.

Total execution time in seconds	
<i>Previous optimization</i>	1.69
<i>New optimization</i>	1.16

Table 2: The performance of the raytracer after the I/O optimization compared to after the Rng optimization.

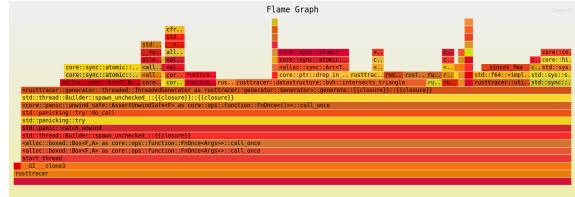


Figure 3: Flamegraph after the Rng improvements.

6 Optimization III

6.1 Description

This optimization is based on the lecture, namely "Lock Contention" and "moving work outside critical section".

The main bottleneck now seemed to be mutex locks. Each thread writes pixel values into a shared Output-Buffer, which requires synchronization. We ended up with kind of combining lock contention with moving calculations outside of loops.

First, We moved camera.width and camera.height outside of the loop, and into variables width and height, and passed them instead. Second, we adjusted the placement of the lock guard so that each row is locked once rather than locking for every single pixel. This was more from testing, and it seems like the overhead of locking was bigger than the effect of locking it every pixel while doing the calculations for each pixel.

6.2 Estimation

This was the most clear bottleneck, but since we didn't find a good way to remove it, we don't think it will make the biggest difference.

6.3 Profile/ Benchmark

Table 3 shows the performance after the Rng improvement and the new lock contention/ moving calculations outside of the loop. It is a small improvement, but nothing remarkable.

Total execution time in seconds	
<i>Previous optimization</i>	1.16
<i>New optimization</i>	1.12

Table 3: The performance after the Rng optimization compared to after the "lock contention".

After applying these changes, the runtime improved from 1.16 seconds to 1.12 seconds, giving a small but measurable speedup.

7 Optimization IV

7.1 Description

This optimization is based on the lecture, that unnecessary allocations and copying should be avoided, especially inside hot loops. When profiling the ray tracer, we noticed a large number of calls to `.clone()`, particularly on the Ray struct inside the intersection code. Every time a ray was cloned, a complete copy of the struct was made, even though most of the time only a reference was needed. Since the intersection functions are executed for almost every ray and bounce, this cloning happened extremely frequently.

By replacing these clones with references (`&Ray`), we removed a large amount of unnecessary memory copying. This change also allowed the compiler to optimize better, because references are small, cheap to pass, and avoid heap allocations entirely.

7.2 Estimation

We think this can improve a lot, since there is a lot of overhead in cloning. This happens inside the intersects, which is the part where the program spends most of its time.

7.3 Profile/ Benchmark

The clone removal changed the time from 1.12s to 0.923s: a 17% improvement.

Total execution time in seconds	
<i>Previous optimization</i>	1.12
<i>New optimization</i>	0.923

Table 4: The performance after the "lock contention" compared to after the memory management, passing references instead of cloning.

The graph below shows the flamegraph after the clone/allocation improvements.

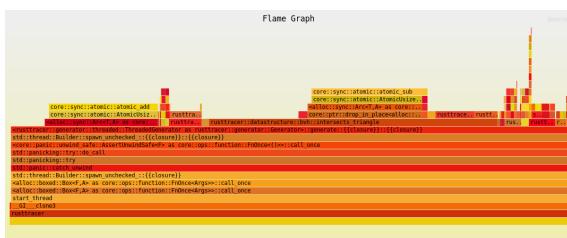


Figure 4: Flamegraph after clone/allocation improvements.

8 Optimization V

8.1 Description

This optimization is based on the lecture's notes about inlining.

Inlining helps with overhead of needing to jump to another section and back. The ray tracer contains many small mathematical functions, especially in vector operations such as dot, normalize, and basic arithmetic operations. Every ray sample uses these operations repeatedly, which is why we used inlining on these

small functions that are frequently used. This embeds the function's body directly into the calling code, removing the call overhead altogether. This will create a larger code, but will also save time.

8.2 Estimation

Inlining removes the cost of jumping to a different memory location and returning back, which saves CPU instructions. Hence, we are expecting a small improvement.

8.3 Profile/ Benchmark

After adding inlining, the runtime improved from 0.923 seconds to 0.884 seconds, about a 4% improvement.

Total execution time in seconds	
<i>Previous optimization</i>	0.923
<i>New optimization</i>	0.884

Table 5: The performance after the reference passing compared to after in-lining.

The flamegraph after these five optimization techniques have been applied is given below.

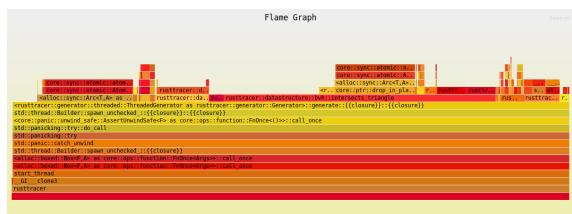


Figure 5: Flamegraph after all 5 optimization technique implementations

9 Conclusion

Wrap up the report in a neat conclusion. Provide the total improvement post all of the optimizations. Mention briefly what ended up performing according to expectation and what did not. Optionally if you want, you can provide future direction you would suggest exploring for additional improvements and why.

The final improvement in the performance of rendering the 50x50 image was 73s into 0.884s, resulting in a 98.8% improvement. The rendering of the 500x500 image took 84s now, almost comparable to the time the baseline took to render the 50x50 image.

The greatest bottleneck is still mutex locking and Arc. We think it could be possible to remove this by better dividing each responsibility of the rows to each thread. This way mutexes could maybe even be removed. Another option could be to send the data over a channel, and then read each row in the main thread. This way you could actually calculate and write the rows in parallel, without worrying about race conditions.

References

- [1] Mara Bos. *Rust Atomics and Locks: Low-Level Concurrency in Practice*. O'Reilly Media, Cambridge, MA, 2023.