Software Systems - Assignment 02 Group 19 - Performance

Zhengtao Huang (5833469, zhengtaohuang) Barry Tee Wei Cong (5662834, btee)

1 Introduction

The purpose of this task is to improve the performance of an extremely slow and inefficient raytracer. Naturally, rays of light are cast from light sources, and they are reflected or refracted before being sensed by the camera. However, ray-tracing implementation is simulated in reverse, where rays of light are cast from the camera and simulated through the scene towards light sources.

The main goal is to modify the source code found in eight subdirectories. The modification of the programme must adhere to a predetermined structure and is not permitted to utilize any external libraries apart from those provided.

In this report, it will consist of at least three performance improvement techniques discussed in class and at least two performance improvement techniques that was done via research. Every performance improvement technique will include a description (what it does and how it works), reasons for speedup, discovery process with expectations, and results after improvement.

2 Experimental Setup

We utilised one of our laptops to benchmark and compare the changes with previous versions. The HP ProBook 440 G7 laptop has an 10th Gen Intel(R) Core i7-10510U CPU. The specifications, that is, the cores, threads, and Random-Access Memory (RAM) of both laptops have been listed in Table 1.

HP ProBook 440 G7	
Physical Cores	4
Logical Cores	8
Total RAM	16 GB
Operating System	Ubuntu Linux 22.04 LTS

Table 1: Laptop Specifications

3 Improvement Techniques

3.1 Minimise IO Overheads by IO Buffering

3.1.1 What It Is & How It Works

IO Buffering is a process used to improve the efficiency of input/output (IO) operations between the process and the storage medium by temporarily storing data in a faster-access memory known as a buffer.

3.1.2 How Does It Speedup Process

IO Buffering speeds up processes by allowing it to minimise physical IO operations by reading and writing data to the memory instead of the physical storage device. The reason why speedup occurs is that the processor need not waste cycles waiting for IO operations to complete and can handle other tasks in the background which is time efficient.

3.1.3 Discovery Process and Expectations

In the Perf flame graph, it shows that the function set_at takes up 93.43% of the overall run-time as per the frame shown in the footnote[1]. By tracing the subdirectories via CLion, the function set_at takes up significant portion of the time because it is called many times in 2 nested for loops found in function generate in generator/threaded.rs. To minimise IO overheads, line 50 of threaded.rs is modified to include let mut buf = BufWriter::new(backup_file); This enables the usage of a dedicated writing buffer to temporarily store the data and speedup the programme.

3.1.4 Results After Improvement

For all results, the first step was to use "cargo build –release" which puts the resulting binary in the directory "./target/release", and the code will run faster compared to compiling in the default debug mode. To obtain the run-time of the raytracer, the "time" command was used. As the code base takes more than an hour to run, the width and height to generate the image have been reduced to 100 x 100 px. The execution time was reduced from 17min 30.451s to 12.541s, giving an improvement of 83x. Subsequently, we reverted the width and height to generate the image to 500 x 500px, which is 6min 13.330s.

3.2 Moving Code Outside Of A Loop

3.2.1 What It Is & How It Works

This technique involves taking instructions that do not need to be repeated in every iteration of the loop and placing them before or after the loop. This reduces the amount of work that the computer has to do in the for loop and makes the programme run faster.

3.2.2 How Does It Speedup Process

Given that the operation of the chosen line of code does not depend on the loop's iteration, the result of that line will produce the same outcome with unnecessary computation resource wasted for every loop ran. Removing it from the for loop ensures that it runs only once, thereby saving time and ensuring efficient usage of processor's resources.

3.2.3 Discovery Process and Expectations

In the Perf flame graph, it reveals the function shade_internal takes up 80.69% of the overall run-time as per the frame shown in the footnote[2]. By tracing the subdirectories via CLion and Perf flame graph, the function raytrace() takes up 98.32% of the overall run-time as it calls shader.shade recursively in the for loop

[1] rusttracer::util::outputbuffer::OutputBuffer::set_at...

[2] rusttracer::shader::mcshader::McShader...

as shown in the footnote[3], which explains the observation of redundant operations tied to shade_internal in the Perf flame graph. First, the team identified line 32 in mstracer.rs: let ray = camera.generate_ray(x as f64, y as f64); to be removed and called once before the loop starts. Next, the team identified line 35 of mstracer.rs, print!(r{x}, {y}); and line 36 stdout().flush().unwrap();, both were removed from the for loop so that it is only called once after the loop ends. To the team's knowledge, print! is an expensive operation especially in repeated cycles.

3.2.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 6min 13.330 to 3min 31.623s, giving an improvement of 1.764x or a reduction of 43.314%.

3.3 Reducing Memory Allocation

3.3.1 What It Is & How It Works

Reducing Memory Allocation is a process to minimise repeated new allocations of memory in a software programme during task operations, enabling efficient usage of memory resources. When Vectors, Hashmaps, Strings, and Box are cloned via .clone() method, a deep copy of an object or data structure is created and invokes copying entire contents into a new memory allocation. To avoid this problem, the programmer can try methods such as avoiding duplicate data, reuse existing objects, borrowing and referencing, or usage of immutable data.

3.3.2 How Does It Speedup Process

Reducing memory allocation speeds up a process naturally reduces its corresponding overheads i.e. data structures that keep track of allocated memory during bookkeeping operations. An overall decrease in memory management would lead to simpler task execution and shorter time to run the task.

3.3.3 Discovery Process and Expectations A

In the Perf flame graph, it reveals the function shade_internal takes up 94.88% of the overall run-time as per the frame shown in the footnote[4]. So, one method with regard to reducing memory allocation was to remove the Box surrounding the Ray, so as to minimise heap memory usage.

3.3.4 Results After Improvement A

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 3min 31.623s to 3min 12.190s, giving a reduction in time of 9.183%.

3.3.5 Discovery Process and Expectations B

Since Ray and Datastructure are used frequently, the team tried to remove the Box surrounding the Datastructure as well.

3.3.6 Results After Improvement B

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 3min 12.190s to 3min 10.600s, giving a time reduction of 0.827%.

3.4 Mitigating Lock Contention

3.4.1 What It Is & How It Works

Mutex Locks are used to ensure data integrity, and lock contention occurs in multithreaded or multiprocessor systems when multiple threads or processes attempt to access a shared resource, with at least one of them requiring write access. When a thread requests a lock and it is already held by another thread, the thread must wait until the lock is released, leading to decreased throughput, increased latency, and potential deadlock situations. To mitigate lock contention, it is important to use finer-grained locks, read/write locks, and minimise the time locks are held.

3.4.2 How Does It Speedup Process

When lock contention is minimised, threads can access shared resources more efficiently, leading to improved throughput and reduced latency. Finer-grained locks and read/write locks allow for more concurrency, enabling multiple threads to access the resource simultaneously under certain conditions. By reducing the time locks are held or eliminating unnecessary locks, the overall system performance is enhanced, less time is spent by threads on waiting and time can be more productive for threads to do useful work.

3.4.3 Discovery Process and Expectations

In the Perf flame graph, it reveals the function shade_internal takes up 95.29% of the overall run-time as per the frame shown in the footnote[5]. So the team looked again at what is the bottleneck for Datastructure and found that Mutex caused lock contention. The process to remove Mutex for Datastructure caused many failed attempts to run the code. With Mutex removed, there was a need to remove the .lock() method for datastructure.lock().unwrap().intersects(ray). The solution was to rewrite the code in mcshader.rs with all the related functions related to datastructure.

3.4.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 3min 10.600s to 2min 42.239s, giving a time reduction of 14.880%.

3.5 Removing Print! and StdOut

3.5.1 What It Is & How It Works

StdOut is a standard output stream for a programme to write its output data. Flush method is used to clear the buffer in the stream, and ensure that all output data is sent out immediately. The Print! function before it and the StdOut operation utilises Central Processing Unit (CPU) resources. In the case of commenting out StdOut operation, there is no need to wait for the programme to output the data to the output destination, which may be extra step from the efficient running of the CPU in running the programme.

[3] rusttracer::raytrace

[4] rusttracer::shader::mcshader::McShader::shade_internal...

[5] rusttracer::shader::mcshader::McShader::shade_internal...

3.5.2 How Does It Speedup Process

Since IO Operation to print and writing to the standard output are operations that are generally orders of magnitude slower than CPU operations, it saves time to comment them out and not spend more time waiting for standard operations to execute. Furthermore, writing to the standard output involves making system calls, which switches user mode to kernel mode, and usually switching incurs substantial overheads as it saves the state from current process and load the state of another process.

3.5.3 Discovery Process and Expectations

In the Perf flame graph, it reveals the function shade_internal takes up 94.76% of the overall run-time as per the frame shown in the footnote[6]. After so much effort to reduce memory allocation and mitigate lock contention, the team decided to employ techniques taught outside of the class. Given that print! and stdout was natural to take up resources after conducting research on it, the team decided to implement a solution by commenting out to see any reduction in runtime while ensuring the output render shows the same raytrace visual image.

3.5.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 2min 42.239s to 2min 29.565s, giving a time reduction of 7.812%.

3.6 Converting f64 to f32

3.6.1 What It Is & How It Works

f64 and f32 are data types used in programming languages to represent floating-point numbers, which are numbers that can represent fractions. f32, also known as float, has 6-9 significant digits of precision, whereas f64, double, has 15-17 significant digits of precision. By using f32 a narrower range and less precise data type, the goal is to not alter the render of the raytrace visual image but minimise intensity of computation by reducing CPU cycles and the memory consumption related to floating-point arithmetic to half as much and hopefully speed up the process.

3.6.2 How Does It Speedup Process

Since a float data type uses half the memory and sometimes half the CPU cycles as compared to a double data type, storing and loading f32 values are considered to be more efficient, especially so in images that contain a large array or matrix of numbers. A float, being smaller, can also have more of them fit into the CPU cache and registers at any point in time, and the improvement in cache utilization leads to fewer cache misses and better overall performance. However, the team does not go that far to say that the given task consumes excessive memory to cause memory-bound workloads.

3.6.3 Discovery Process and Expectations

The team came across an article called "Rust: f64 vs. f32" that was dated in 2017. Mr Hugo tested and found 30% improvement in run-time. This prompted the team to execute the same strategy and determine whether this optimization technique is useful for future projects or tiny embedded processors found on drones.

3.6.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 2min 29.565s to 2min 26.016s, giving a time reduction of 2.373%. Given that the improvement is only a mere 2.373%, the team can deduce that modern day CPU and memory found in laptops are more advanced to handle computation using double-precision datatypes. The optimization may work better when implemented on small drone flight controllers.

3.7 Dig Deeper into Code Libraries - The Random Number Generator

3.7.1 What It Is & How It Works

Using codes from Libraries, be it from official or non-official sources, are efficient to achieve the intended function as set in project requirements but may be deemed as black-box or turn-key solutions given that the code may be too huge to fully understand everything. With such usage, this may produce undesirable side effects in the testing stage. One way is to read up on documentation or check on feedback after using the library and making according changes.

3.7.2 How Does It Speedup Process

Given that the random number generator is relied on for random sampling to determine light distribution in ray-tracing projects, it can be logical to deduce that this library will be called a significant number of times to simulate soft shadows, depth of field, and even path tracing. Hence, changing the random number generator for the purpose of speed while retaining the same function is possible.

3.7.3 Discovery Process and Expectations

The world of image processing academics and professionals are making constant comparisons when it comes to utilisation of various code libraries. As such, the team went to search on the drawbacks of OsRng random number generator. It revealed that OsRng is typically used to achieve Cryptographic Security in industries like finance and government institutions needing to ensure data confidentiality. Hence, the security feature is not useful and causes the system to be slower as it had to make system calls. As such, ThreadRng was a simple and direct replacement for fast performance in the ray-tracing project.

3.7.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 2min 26.016s to 1min 35.654, giving a time reduction of 34.491%.

3.8 Mitigating Lock Contention - Follow-Up from Section 3.4

3.8.1 Discovery Process and Expectations

In Section 3.4, the program was improved by removing unnecessary Mutexs. In this section, the team

[6] rusttracer::shader::mcshader::McShader::shade_internal...

has shown how to mitigate lock contention by reducing the amount of code in each critical section. In src/generator/threaded.rs, the team realized that callback(x,y) is used by a parent function that allows passing of a function as an argument to another function. Callback functions are usually used in asynchronous programming, and they can be ran by multiple threads. Hence, it is more efficient to call the callback function that obtains the results with parallised computing taking place prior to locking it with the guard variable. Previously, it was slow because the guard was locking the callback function, preventing it from being scheduled to run in parallel.

3.8.2 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 1min 35.654s to 34.662s, giving a time reduction of 63.763%.

3.9 Removing Unnecessary Atomic Reference Counting

3.9.1 What It Is & How It Works

Atomic reference counting, or Arc, keeps a record of the reference counts of a piece of data stored on the heap. It ensures thread safety by using atomic operations for reference counting. An Arc is often used with a Mutex to avoid data race when multiple threads need to write to the same piece of memory. However, Arc is more expensive than normal memory accesses. When used when reference counting is not necessary, it creates unnecessary overheads in the programme. In the code base for the ray-tracer, the team has found multiple usages of unnecessary Arc, and the code is optimised by removing such usages of Arc.

3.9.2 How Does It Speedup Process

The team has removed several usages of Arc where it is not necessary. One of the most important of them is the usage of Arc to the "DataStructure" object. Instead, we used "Box" to put the "DataStructure" on the heap and changed the cloning of Arc pointers into references to the "DataStructure". In this way, unnecessary overheads caused by redundant usage of Arc are eliminated.

3.9.3 Discovery Process and Expectations

The team looked into the code and found that the use of Arc on "DataStructure" is unnecessary, as it is not accompanied by the use of Mutex, and there is no data race associated "DataStructure". We expected that this change would speed up the programme significantly, because the type "DataStructure" is used by the shader. As previous profiling results have shown, the shader function takes up a huge amount of percentage of the execution time of the programme.

3.9.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 34.662s to 30.844s, giving a time reduction of 11.015%.

3.10 Improved Parallelism With More Cores and Threads

3.10.1 What It Is & How It Works

A computing system with more CPU cores and better micro-architecture design would increase the performance of this programme significantly. As the number of CPU cores and threads increases, there are more paths for the operation to be split up for multithreaded applications. Given that each core can independently execute instructions and the multithreaded application can take advantage of multiple cores for parallel processing. Furthermore, such applications also take advantage of multiple threads where they can distribute the workload across cores.

3.10.2 How Does It Speedup Process

Amdahl's Law is a formula that helps explain how the addition of more cores and threads affects the overall speedup of a process. It provides insight into the potential limitations of parallel processing. Amdahl's Law states that the speedup achieved by parallelizing a task is limited by the portion of the task that cannot be parallelized. This non-parallelizable portion is represented by the variable "S" in the formula. The formula is as follows:

$$S = \frac{1}{(1-f)+\frac{f}{n}}$$

newline "Speed-Up" represents the improvement in processing time that parallelizes the task. "S" represents the proportion of the task that cannot be parallelised. "N" represents the number of cores or threads used for parallel execution.

3.10.3 Discovery Process and Expectations

The HP ProBook 440 G7 mentioned in Sextion 2 was bought in 2020, and a new laptop was bought in 2023 to perform more computationally intensive tasks. The newer HP ZBook Power uses a 13th-Generation Intel(R) i7-13700H CPU that has 14 cores and 20 logical cores running on Windows 11. The newer laptop allows more parallelism and has a better cooling system that keeps the performance high.

3.10.4 Results After Improvement

For the width and height of the generated image set to 500 x 500px, the execution time decreased from 30.844s to 12.672s, giving a time reduction of 58.916%.

4 Results - Summary

This section of the report shows a complete summary of all the benchmarked timings from every optimization. In Table 2, a summary is shown. Given that the original default code could not even run, the pixel count was set at 100 x 100px, and the improvement from base to Optimization 1 was 988% or a speedup of 83x. From Optimization 1 to Optimization 10, the time reduced from 6min 13.330s to 12.672s which gives an improvement of 96.606% or a speedup of 29x.

Table 2: Summary of All Execution Times

Opt.	Pixel	Time	% Reduced
Base	100 x 100	17min 30.451s	
1	100 x 100	12.541s	988.061
1	500 x 500	6min 13.330s	
2	500 x 500	3min 31.623s	43.314
3A	500 x 500	3min 12.190s	9.183
3B	500 x 500	3min 10.600s	0.827
4	500 x 500	2min 42.239s	14.880
5	500 x 500	2min 29.565s	7.812
6	500 x 500	2min 26.016s	2.373
7	500 x 500	1min 35.654s	34.491
8	500 x 500	34.662s	63.763
9	500 x 500	30.844s	11.015
10	500 x 500	12.672s	58.916

5 Performance - Criterion Benchmark

As required, the team performed a benchmark with Criterion to evaluate the optimised version of the code. We wrote a benchmark with Criterion and performed the benchmarks with the "cargo bench" command. We compared the results from the baseline and the latest version of the code. The benchmark is performed on the HP ProBook mentioned in Section 2. Considering that the baseline version is extremely slow, the sample size is configured to 10, and the size of the image is configured to 100 times 100 pixels. Results are listed in Table 3. Based on the results, we may estimate that the optimised version of the programme achieves a speedup of 775 times.

Table 3: Results from Criterion Benchmarks

	Baseline	Final Version
Image Size	100 x 100	100 x 100
Confidence Interval	90%	90%
Lower Bound	866.74 sec	1.1327 sec
Upper Bound	892.78 sec	1.1704 sec
Best Estimate	892.78 sec	1.1506 sec

6 Conclusion

In conclusion, a thorough study on optimization techniques was done to improve performance run-time, mainly 4 techniques from the lecture and 5 other techniques that was researched by the team. From the start without any optimization, the programme improved by 775x and the team members believe that this skill set of optimizing programmes is a useful skill in the workplace after graduation.

References

[1] Rust
$nnethercote.github.io.https//nnethercote.github.io/perf-book/io.html : text = Buffering$
"I/O - The Rust Performance Book," (accessed Nov. 27, 2023)

[2] J. Donszelmann and V. Roest
$https : //cese.pages.ewi.tudelft.nl/software - systems/slides$
"Software Systems Lecture 2," Slides - Software Systems, Nov. 24, 2023. (accessed Nov. 27, 2023).

[3] Rust
$web.mit.edu.https : //web.mit.edu/rust - lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/second-edition/ch12-06-writing-to-stderr-instead-of-stdout.html$
"Writing Error Messages to Standard Error Instead of Standard Output - The Rust Programming Language," (accessed Nov. 27, 2023).

[4] H. Tunius
$https : //hugotunius.se/2017/12/04/rust - f64 - vs - f32.html$
"Rust: f64 vs f32," hugotunius.se, Dec. 04, 2017. (accessed November 27, 2023)

[5] bk2204
$https : //stackoverflow.com/questions/71048413/rng-getting-randomness-from-operating-system-vs-crypto-rngs$ "Rng getting randomness from operating system vs crypto rng's," Stack Overflow, Feb. 09, 2022. (accessed Nov. 27, 2023).