Software Systems - Performance Assignment

Jiacong Li (5656354, jiacongli@tudelft.nl)   Tongyun Yang (5651794, tongyunyang@tudelft.nl)

1 Description

As mentioned in [1], the assignment is about improving the performance of a given raytracer mostly with techniques discussed during the lecture. Unlike the real world, ray tracing does not simulate rays of light from light sources, they do the opposite instead. In [2], it metaphorized this process as "Tracing the rays from eyes" which is very appropriate. The source code of the raytracer provided mainly consists of 8 sub-directories, and improvements should be applied to them. However, due to the fact that the sub-directory - "datastructure" is reasonably efficient, it should be focused on less. The program should not use external libraries without TA's permission.

A record of all improvements applied to the program is made as well as a checklist for all requirements. They are delivered in the git repository on Gitlab[1] in the README.md file on the "2-performance" branch. The record shows the sequence of improvements made, and the checklist directly displays what is being delivered in the assignment. Note that the record of improvements in the README.md file is displayed in the same order as in this report in section 3. However, the README.md is more concise and lack discussions, hence refer to this report for more details.

This report is as important as README, they do not contain the same contents. Hence both worth reading.

Due to limited space, clear abbreviations are used in this report, e.g. "Exe. Time" referring to "Execution Time", "Perf." referring to "Performace", "Sam. Rate" referring to "Sampling Rate" and so on.

2 Analysis Setup

After implementing the improvements, the performance analysis is done respectively. Only one testing platforms is used, it is displayed below:

- Dell G15 5511
  - 8 Cores 16 Threads Intel i7
  - 16 GB RAM
  - Ubuntu 22.04 LTS

The number of cores and threads supported of the laptop would make a the execution time of the program different. While running the program, it is observed that the program does not run "smoothly" sometimes, hence, an immature guess is that the size of RAM also affects the speed of the program due to the reason that data would be written to RAM while executing the program.

All naive benchmarks below run the same program with "**time cargo run –release**". Naive benchmarks are done ten times (result for the first run is expelled) and averaged to get a clearer result. The result of naive benchmarks will be displayed in section 3 together with all optimizations delivered.

Note that the "mature" benchmark using criterion is displayed in the section 4. The result of naive benchmarks is completely different from the "mature" one due to the fact that the later ones executes very slow, hence the size of the figure is rather small.

3 Optimizations

In this section, optimizations done for the program are presented in the sequence they are applied. In each subsection, the optimization is described together with the discovery of it. The effect of the optimization is displayed, and the reason of it affecting the performance is explained. Hence, there will be three sub-subsections, each for a topic above respectively.

3.1 set_at() in util/outputbuffer.rs

3.1.1 Description & Discovery After cloning the git repository, the profiler is directly run for the program. It is shown clearly in figure 1 that function set_at() occupies 87.57% of all the running time. Note that due to the very long time the program runs with a rather big figure, the size of the figure generated is reduced to "Height: 100, Width: 100, Sampling Rate: 100".

Function set_at() is called by function generate() in generator/threaded.rs. It sets the color of a pixel, then writes the RGB value into a file in 2 nested for-loops.

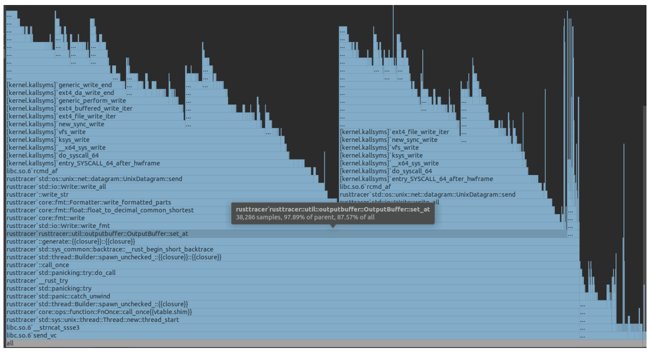


Figure 1: Function set_at() profiler

3.1.2 Optimization & Result The function is improved by removing the for-loops due to redundancy

[1]A DevOps software package that combines the ability to develop, secure, and operate software in a single application.

	Fig. Size	Sam. Rate	Exe. Time	Perf.
Base.	100×100	100	10min 07s	/
Opt.3.1	100×100	100	4.5s	×134.89
Opt.3.1	500×500	200	3min 41s	/

Table 1: Effect of set_at() Optimization

which is discussed further in the "Explanation & Discussion" section. The functionality of writing RGB values to the backup file is not changed. However, it now writes the RGB value of the pixel which is executing on at present to the file using "BufWriter".

The result is displayed in table 1. The optimization speeds the program up to 134.89 times faster, and when the profiler runs once again. The set_at() function occupies too little portion of time that it cannot be seen anymore.

3.1.3 Explanation & Discussion It could be directly spotted that the nested for-loops are redundant, due to the fact that it is writing the RGB value of all pixels each time when the color of one single pixel is set. Furthermore, when writing the RGB values to the backup file, a "BufWriter" replaces the original buffer.

This optimization mainly concerns **Moving operations outside a loop** and **I/O buffering**.

3.2 intersects_triangle() in datastructure/bvh/-mod.rs

3.2.1 Description & Discovery The profiler runs again after the first optimization, and it is spotted that intersects_triangle() takes up a lot of the execution time. The observation could be clearly seen in figure 2. Figure 3 shows that a() in scene/triangle.rs takes up most of the time among all functions called by intersects_triangle(). The reason that intersects_internal() is not optimized first is due to the fact that it belongs to "datastructure", and it is efficient enough by default.

Function intersects_triangle() is only called in intersects_internal(). The functionality of it is to return the intersection point between a ray and a triangle if it exists, and "None" otherwise.

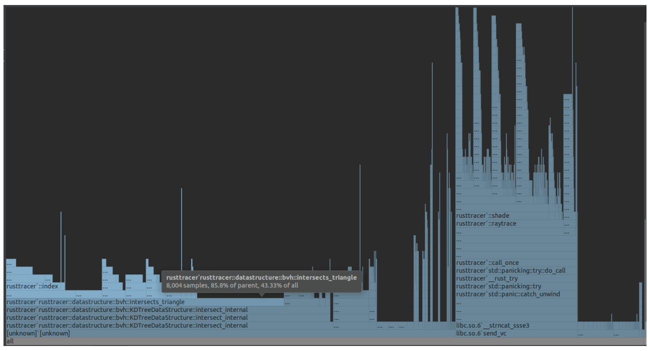


Figure 2: Function intersects_triangle() profiler 1

3.2.2 Optimization & Result "#[inline]" are added before a(), b() and c() for inlining. Note that dot() and cross() are also optimized, but they will be mentioned in the further section 3.3 together with other functions.

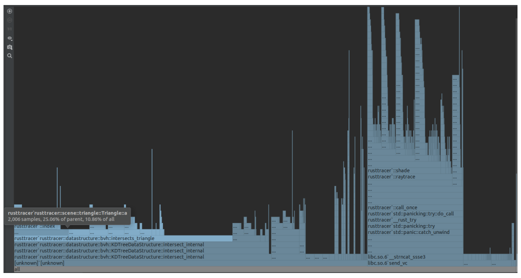


Figure 3: Function intersects_triangle() profiler 2

	Fig. Size	Sam. Rate	Exe. Time	Perf.
Opt.3.1	500×500	200	3min 41s	/
Opt.3.2	500×500	200	3min 38s	3s faster

Table 2: Effect of intersects_triangle() Optimization

The result is displayed in table 2. Applying inlining gives a speedup of approximately 3 seconds.

3.2.3 Explanation & Discussion In intersects_triangle(), it is observed that a(), b(), c(), dot() and cross() are called quite often. Their functionality is to do certain calculations. Hence, inlining is thought of and applied.

This optimization mainly concerns **Inlining**.

3.3 Pass by reference 1

3.3.1 Description & Discovery As mentioned above in the previous section 3.2, dot(), cross() are spotted. The original code contains a lot of clone(), and hence an idea of getting rid of clone() appeared. This includes dot, cross() and intersects_triangle().

3.3.2 Optimization & Result In dot() and corss(), "other" is now passing by reference. Meanwhile in intersects_triangle(), both "ray" and "triangle" are now passing by reference too.

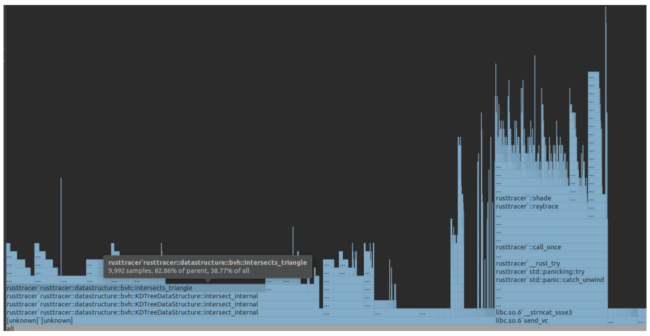


Figure 4: Pass by reference 1 profiler before Opt.

The result could be seen in the comparison between figure 4 and 5. Function intersects_triangle() occupies less percentage in the flame graph of the whole program. In table 3, it is clearly shown that 2 seconds of speedup is gained.

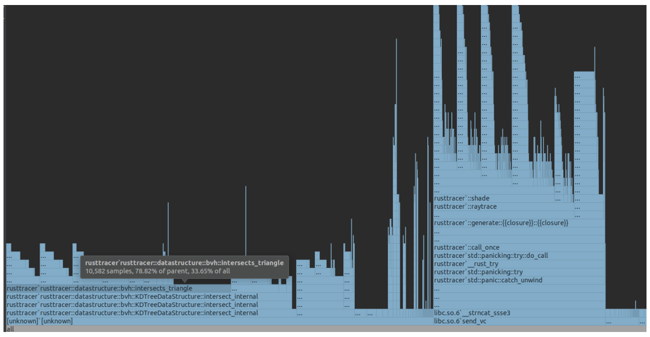


Figure 5: Pass by reference 1 profiler after Opt.

	Configuration	Exe. Time	Perf.
Opt.3.2	Same[2]	3min 38s	/
Opt.3.3	Same	3min 36s	2s faster

Table 3: Effect of Pass by Reference 1 Optimization

3.3.3 Explanation & Discussion It is discussed during the lecture that, cloning vectors would consume time. That is the motivation of adjusting the functions to passing parameters by reference. Note that during optimization, it is aware that cloning "Arc" type in not time consuming. However, parameter "triangle" (type of Arc) is also adjusted, and it will remain to be passed by reference throughout the rest of the optimizations.

This optimization mainly concerns **Static allocation / reducing allocation**.

3.4 raytrace() in raytracer/mstracer.rs 1

3.4.1 Description & Discovery According to previous flame graphs displayed. Figure 4 and figure 5 shows that there are two recursive calls that take up most of the time - intersect_internal() and shade_internal(). Function raytrace() if one of the functions that calls shade_internal() indirectly. It is observed that there are redundancy occurring in the function.

raytrace() is called in generate_internal(). It generates a ray (vector) associated with a coordinate of x and y which it takes in.

3.4.2 Optimization & Result All redundant executions inside the for-loop are move out. This leaves only one line of code left inside the for-loop. The division on parameter "out", the print statement and the camera.generate_ray() are all moved out.

From table 4, it shows clearly that this optimization gives huge speedup of 58 seconds.

3.4.3 Explanation & Discussion The reason all the functions could be moved out is due to redundancy - they only need to be execute once when raytrace() is called. They are all dependent on parameters that will stay stable within one call of the function. This is the reason why the program speedup.

	Configuration	Exe. Time	Perf.
Opt.3.3	Same	3min 36s	/
Opt.3.4	Same	2min 38s	58s faster

Table 4: Effect of raytrace() in raytracer/mstracer.rs 1 Optimization

	Configuration	Exe. Time	Perf.
Opt.3.4	Same	2min 36s	/
Opt.3.5	Same	2min 22s	14s faster

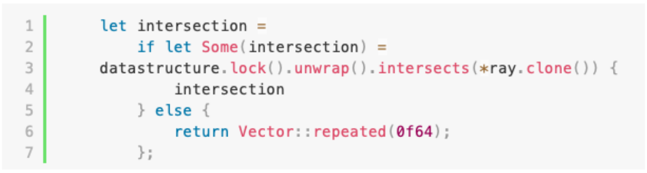
Table 5: Effect of raytrace() in raytracer/mstracer.rs 2 Optimization

This optimization mainly concerns **Moving operations outside a loop**.

3.5 raytrace() in raytracer/mstracer.rs 2

3.5.1 Description & Discovery Following the same story as mentioned in section 3.4. The focus is on the same function - shade_internal(). Through observation, the very first if-else statement which is shown below in figure 6 could be moved out of the function into raytrace().

Secondly, it is figured that parameter "datastructure" is always passed by "Arc(Mutex(Box(dyn DataStructure)))", and type "Mutex" and "Box" could be removed.



```
    let intersection =
        if let Some(intersection) =
    datastructure.lock().unwrap().intersects(*ray.clone()) {
            intersection
        } else {
            return Vector::repeated(0f64);
        };
```

Figure 6: The very first if-else statement in shade_internal()

3.5.2 Optimization & Result Moving the if-else statement out of shade_internal() requires the function to receive and extra "Option" parameter. The adjustments are made for raytracer(), shade() and shade_internal(). As for the removal of type "Mutex" of "datastructure", some files needs to adjusted as well. Refer to the code (or README.md) for adjustments in detail.

From table 5 it is shown clearly that it gives a speedup of 14 seconds.

3.5.3 Explanation & Discussion The reason why if-else statement could be moved is as follows: shade_internal() is called by shade(), shade is called by raytrace() in a for-loop, and shade_internal() would also call itself recursively. Hence moving the if-else statement out could save time executing intersects() and probably save time from branch prediction as well.

The reason why "datastructure" does not need to be type "Arc(Mutex(Box(dyn DataStructure)))" is because the value of "datastructure" never changed throughout the program and it is only about reading it. Locking it each time for access is a waste of time.

This optimization mainly concerns **Moving operations outside a loop, Lock contention and prob-**

	Configuration	Exe. Time	Perf.
Opt.3.5	Same	2min 22s	/
Opt.3.6	Same	37.6s	×3.78

Table 6: Effect of generate() in generator/threaded.rs Optimization

	Configuration	Exe. Time	Perf.
Opt.3.6	Same	37.6s	/
Opt.3.7	Same	34.3s	3.3s faster

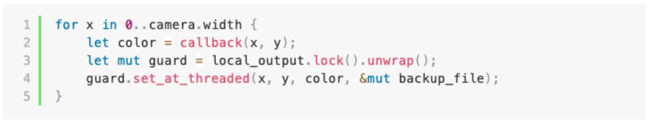
Table 7: Effect of new() in scene/texture/mod.rs Optimization

ably Branch prediction (Not sure if branch prediction is actually improved).

3.6 generate() in generator/threaded.rs

3.6.1 Description & Discovery Following the story in section 3.5. The function which is optimized in the very beginning is thought of. The lock contention is being removed. The color is now set with callback() before local_output() being locked.

3.6.2 Optimization & Result The adjusted part in generate() is as shown in figure 7.



```
for x in 0..camera.width {
    let color = callback(x, y);
    let mut guard = local_output.lock().unwrap();
    guard.set_at_threaded(x, y, color, &mut backup_file);
}
```

Figure 7: The adjustment made to function generate()

In table 6, it is clear that the optimization gives about 3.78 times speedup.

3.6.3 Explanation & Discussion In generate() where set_at() is called, there is lock contention. Executing callback() would make all the other threads stall before the lock(), and hence it is a waste of time.

This optimization mainly concerns **Lock contention**.

3.7 new() in scene/texture/mod.rs

3.7.1 Description & Discovery This function is found when every single file is being analyzed for improvement carefully. The profiler does not provide useful information any longer. Similar to the first optimization made, this optimization is regarding the use of "I/O buffering".

3.7.2 Optimization & Result A "BufReader" is implemented to read the file instead of reading directly from the file each time.

According to table 7, the program gains a 3.3 seconds speedup.

3.7.3 Explanation & Discussion It is discussed in class that "BufReader" would be faster, hence it is applied.

This optimization mainly concerns **I/O buffering**.

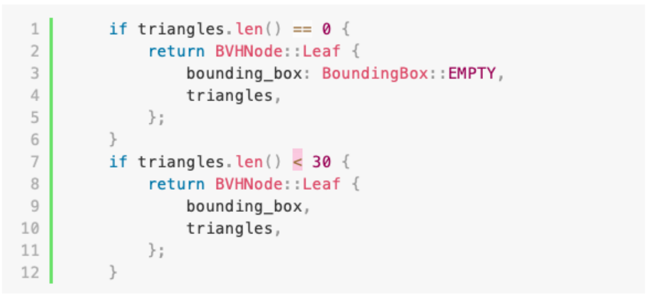
	Configuration	Exe. Time	Perf.
Opt.3.7	Same	34.3s	/
Opt.3.8	Same	23.2s	×1.47

Table 8: Effect of new_internal() in datastructure/bvh/node.rs Optimization

3.8 new_internal() in datastructure/bvh/node.rs

3.8.1 Description & Discovery After implementing the optimization in section 3.5. The same pattern is followed to find more similar potential improvements. The grep[3] function implemented in the last assignment is used to find similar patterns. Keyword "return" is searched with grep, and then all the results are analyzed. Function new_internal() is spotted and adjusted.

3.8.2 Optimization & Result The if-else statement in figure 8 is moved out of the function into the new() function which is in the same file above it.



```
    if triangles.len() == 0 {
        return BVHNode::Leaf {
            bounding_box: BoundingBox::EMPTY,
            triangles,
        };
    }
    if triangles.len() < 30 {
        return BVHNode::Leaf {
            bounding_box,
            triangles,
        };
    }
```

Figure 8: The adjustment made to function new_internal()

According to table 8, this optimization gives a speedup of 1.47 times.

3.8.3 Explanation & Discussion There is an doubt on this optimization. new_internal() is a function that would recursively call itself, and hence removing the if-else statement at the very first might change the behaviour potentially. However, the output of the program does not seem to be affected. Hence this adjustment is being kept throughout further optimizations.

This optimization mainly concerns **Moving operations outside a loop**.

3.9 Random Generator

3.9.1 Description & Discovery It is always aware that the random generator takes up part of the execution time. However, whenever adjustments are planned to make on it. A documented sentence - "// Best random distribution" is seen in util/mod.rs. After applying a lot optimizations and due to the fact that the deadline of the assignment is approaching. We finally tried some adjustments with the random generator. Two different random generators are tested.

[3]grep is a command-line utility for searching plain-text data sets for lines that match a regular expression.

	Configuration	Exe. Time	Perf.
Opt.3.8	Same	23.2s	/
Opt.3.9	Same	16.7s	×1.39

Table 9: Effect of Random Generator Optimization

3.9.2 Optimization & Result Both thread_rng() and fastrand gives exactly the same speedup, hence, only results using thread_rng() would be displayed.

According to table 9, the speedup of changing the random generator into thread_rng() gives a speedup of 1.39 times.

3.9.3 Explanation & Discussion Adjusting the random generator into thread_rng() does deliver faster speed, but sacrifices security. Function thread_rng() is fine for this assignment, but not suitable for cryptography related programs.

This optimization mainly concerns **Using functions that are more appropriate for the program**.

3.10 The Rest

3.10.1 Description & Discovery Besides all the optimizations mentioned above, there is one optimization that is made but does not give speedup at all. However, we personally think that the optimization is still valid because it manages to get rid of a lot of clone(). Passing "generator", "raytracer" and "shader" by reference instead of cloning everywhere.

3.10.2 Optimization & Result This optimization starts at rederer/mod.rs. Removing clone() for the three parameters, then adding "&" to where it is defined, the compiler states that lifetime should be specified. Hence we do what the compiler says till there is no more error.

It is aware that cloning type "Arc" is not time consuming before applying this optimization. However, it just feels that something is wrong when seeing clone() everywhere. Hence we gave the optimization a try.

To our expectation, no speedup is gain from the optimization.

3.10.3 Explanation & Discussion The explanation to the result is that cloning type "Arc" is not time consuming. However, by applying this optimization, most of the clone() are gotten rid of.

4 Criterion Benchmark

The "mature" benchmark is run with command "**cargo bench**". Note that a separate sub-directory is made for running benchmark - "benches". In order to get **bonus points**, we made several different benchmarks using criterion. The first one is a benchmark only for function set_at(), the rest are the benchmarks for each optimization against the last optimization. They will be delivered respectively in two tables. Refer to the code on Gitlab for the implementation of benchmark in detail, and detailed screenshots can be viewed in README.md. All benchmarks using criterion use the setting of "Graph size: 25 × 25 Sample rate: 25".

	Exe. Time	Improvement Perf.
Before Opt.	51.55ms	/
After Opt.	130.05us	99.75%

Table 10: set_at() benchmark using criterion

	Exe. Time	Imp Perf.
Opt.3.1	75.207ms	97.15%
Opt.3.2	72.051ms	1.16%
Opt.3.3	70.965ms	2.37%
Opt.3.4	52.109ms	24.96%
Opt.3.5	50.005ms	9.39%
Opt.3.6	15.177ms	71.07%
Opt.3.7	14.601ms	3.41%
Opt.3.8	13.877ms	4.39%
Opt.3.9	10.609ms	23.64

Table 11: Benchmark against Baseline of every optimization using criterion

5 Discussion

Most of the optimizations contribute to the speedup. However, some optimizations are achieved due to luck, e.g. optimization in 3.8. The fact is obvious that the optimization is valid, however, there might be potential risk of the behavior becoming weird under other circumstances. Further analysis is required for this assignment. The result we are delivering is definitely not the limit of the performance of the program.

6 Conclusion

As shown from all the results that we delivered, the speedup of the program is approximately 134.89 × 13.23 = 1784.5947 times. **All the baseline requirements as well as the requirements for bonus points are met**. It is confident that the program performs (regarding only time) similarly as the program that it is going to be tested against.

References

[1] "Software systems," Technische Universiteit Delft, (Date last accessed 11-28-2022). [Online]. Available: https://software-fundamentals.pages.ewi.tudelft.nl/software-systems/website/index.html

[2] "Ray tracing (graphics)," From Wikipedia, the free encyclopedia, (Date last accessed 11-18-2022). [Online]. Available: https://en.wikipedia.org/wiki/Ray_tracing_(graphics)