UNIVERSITY OF AMSTERDAM

PERFORMANCE ENGINEERING PROJECT

Optimizing "Raytracing in One Weekend"

June 7, 2022

Students:
Kevin Kuurman
11011017

Chaitanya Kumar
13821369

Daniël Boelman
12258490

Dimitri Hooftman
13246038

Group:
Group 5

Course:
Performance Engineering

1 Introduction

A modern computer is able to push millions of 3D polygons a second to project a three-dimensional scene to a two-dimensional image. Due to the nature of this process it is very hard to make these images look realistic. This is because shadows, reflections, refractions, transparency are not simple surface level textures on a plane. To simulate this modern day graphics engines use complicated techniques such as shadow casting where a scene is rendered multiple times and stored to a temporary buffer to simulate as though an actual ray has been cast.

Ray tracing takes more after how light works in the physical world, where rays of light bounce around and eventually reach your eye which allows you to perceive the world. This process however, while simpler on paper, is much more straining on hardware to perform. The first accelerators with ray tracing have only recently been released, and before that it was either emulated in other distributed or parallel technologies or plainly performed by the CPU.

In section section 2 we give a brief overview of the baseline source code and comment on its runtime profile. In section 3 we outline our performance goals based on our analysis of the baseline code and its runtime profile, present a high-level model of the application, and expound on the different optimizations.

2 The Baseline Source Code

Many results come up when scouring the web for implementations of a ray tracer. We choose the implementation offered by Dr. Peter Shirley's 'Ray-tracing in One Weekend' due to the fact that it is accompanied by a comprehensive book explaining the concept and process of ray-tracing.

It should be noted that the books and the accompanying source code form a pedagogical project and are excellent resources for learning the underlying physical and mathematical concepts for ray-tracing. The changes that we make to this code over the course of this project should not be misconstrued as critiques of the software engineering or performance properties of the source code.

2.1 Analysis of the Code

The supplied code is in C++11 and consists of a comprehensive framework to define shapes, cast rays and project scenes to an image and the program is sequential. An overview of the algorithm is given in algorithm 1.

Algorithm 1 High-level overview of the ray-tracer application

```
    Input: image resolution $(M, N)$ (pixels), samples-per-pixel $S$
 1: world ← cornell_box()  ▷ Initialise geometry primitives, create materials, allocate materials
    to primitives
 2: for $i$ in $0..M$ do
 3:     for $j$ in $0..N$ do
 4:         ray ← get_ray(M,N)        ▷ Construct ray corresponding to current pixel coordinates
 5:         pixel_color ← 0x000000
 6:         for $s$ in $0..S$ do
 7:             color += ray_color(ray, world)
 8:         end for
 9:         write_color(color, i, j)                          ▷ Store color value in image buffer
10:     end for
11: end for
```

The procedure in line 7 is the main work-horse of the ray tracer; it determines the colour of the pixel associated with the corresponding ray of light. This is further broken down into two parts.

1. Determine which part of the world (geometry) the ray intersects with. If the ray misses all geometry, then the background colour (black) is returned.

2. If an intersection point is found, use the material properties and the surface normal of the object to calculate the scatter and color of the ray.

The reflection is implemented by recursively calling `ray_color` again with the reflected ray as its argument. It also takes a recursion depth threshold as one of its arguments, which has been elided in algorithm 1 for brevity as we do not change it from its default value as part of our analysis.

For the purpose of performance engineering, we only concern ourselves with the loops starting from line 2. We also do not time the I/O, i.e. formatting the colour values into an image file format and writing it to the filesystem. The computed colour components are stored as double-precision IEEE 754 floats into a contiguous buffer in the inner loop (line 3).

2.2 Relevant Implementation Details

The first rule of performance engineering is to measure what you optimise, and the first step of measurement is to generate a runtime profile of the application. In this section we present a high-level overview of some of the important subroutines and classes of the application.

Computational Geometric Objects and their Containers

The geometric entities of the world are implemented as specialisations of the `hittable` abstract class that exposes a virtual method `hit` that returns a Boolean, takes as arguments a ray and a non-const reference to a hit record struct and does two things:

1. Determines if the ray intersects the object. Returns false if it does not.

2. If intersection is found, it calculates the surface normal, coordinates of the point of intersection, and stores these along with a pointer to the material of the object.

A `hittable_list` is another specialisation of `hittable` that serves as a contiguous container of pointers to other `hittables`. It specializes the hit method by iterating over its list of `hittable` pointers and calls their hit methods. Doing so requires the creation of a temporary hit record struct that at the end of all iterations stores details from the point of intersection closest to the origin of the ray. This is the type of `world` in line 1 in algorithm 1. The box primitive is also implemented as a hittable list of rectangles, where the rectangles form the walls of the box.

Therefore, `hittable_list::hit` is called at the top level inside `ray_color`, which in turn calls the `hit` methods for all objects in the world. Let N_h be the number of objects in the world, and let d be the maximum number of reflections (i.e. maximum recursion depth). Then `hit` is called $\leq d \times N_h$ times per call to `ray_color`. In the cornell box scene that we focus on, $N_h = 14$ and $d = 50$, so that $d \cdot N_h = 700$. So, for an image of height 600 pixels and aspect ratio $1 : 1$, all the primitive hit methods are called $\leq d \cdot N_h \cdot S \cdot M \cdot N = 70 \cdot 50 \cdot 3600 = 1.638 \cdot 10^8$ times. We anticipated that the runtime profile of the baseline application will be dominated by the different `hit` methods.

Now, since the box object is implemented as `hittable_list` of six rectangles, the ray-box intersection will call the ray-rectangle intersection for each of its six faces. Therefore, we need to keep in mind that a significant fraction of the rectangle hit calls in the profile will come from box hits. We can use the fact that each box hit calls six rectangle hits to estimate the actual contribution of rectangle hits.

2.3 Runtime Profile

We used gprof to generate a runtime profile of the entire application. gprof is a sampling profiler that can provide accurate estimates of the number of times each top-level function is called. Because of the recursive nature of the pixel colouring subroutine and the non-uniform recursion-depth, visual representations of the call-graph (e.g. flamegraphs) were too dense and non-uniform to be helpful. Therefore, we mostly inspected flat profiles to find hot paths. The function-wise distrbution of runtime contributions, along with total function calls, is shown in fig. 1.

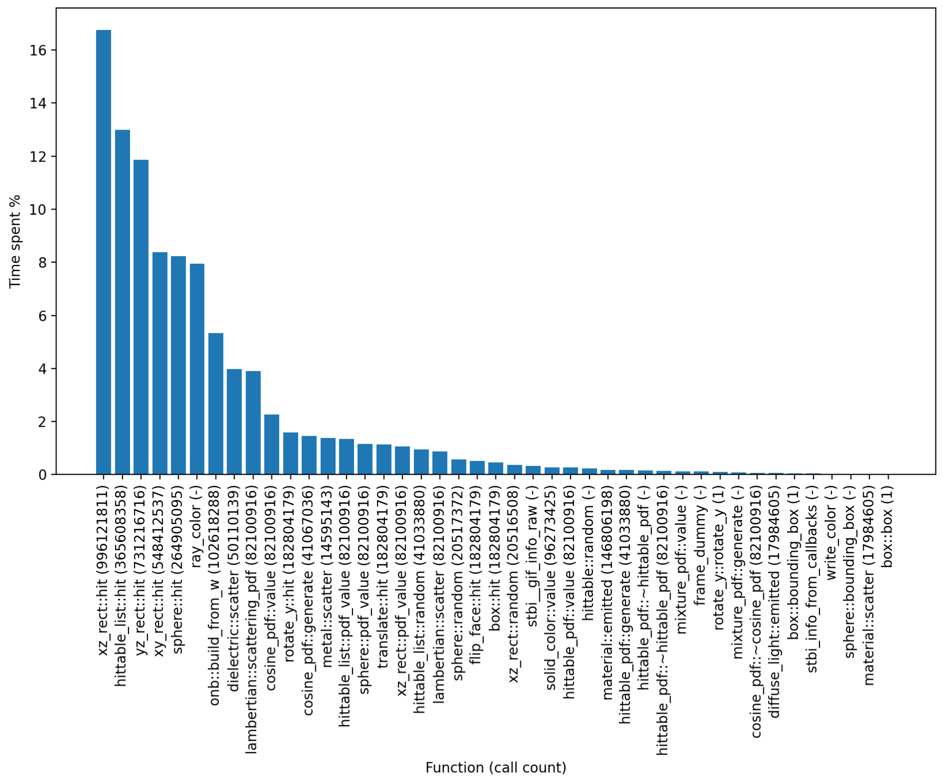


Figure 1: *Percentage of total time spend per function* – flat profile – no call-graph information. *Shown on the xtics are function names along with the number of times they are called throughout the program runtime.*

We note a few things here.

1. Three of the top-five most time-consuming functions are the ray-rectangle intersection functions. Most of the hot functions are the ray-object intersection functions.

2. While `box::hit` looks innocuous sitting in the tail-end of the distribution, recall that we mentioned in section 2.2 that this baseline method calls six of the rect-hit methods. Consequently, about a billion of the rect-hit function calls are contributed by `box::hit`.

By parsing and filtering the gprof output in fig. 1, we calculated that $\approx 48\%$ of the `.._rect::hit` calls were contributed by `box::hit`, which means that `box::hit` contributes at least 17% to the total runtime. Note that this is a percentage of the entire program runtime, i.e. including things like scene-setup and file I/O, which we do not measure. Therefore, its contribution to the actual ray-tracing kernel would be somewhat larger than 17% – this figure is just the lower bound.

3 Performance Engineering

Given the understanding of the application as explained in section 2.2 and its runtime profile in section 2.3, we can perform some preliminary analysis and propose some hypotheses.

3.1 Performance Goals

Given the complexity of the application, we found it non-trivial to gauge what a reasonable "hard number" for target speedup should be. We had two goals at a very high level.

1. Parallelize the code and achieve linear scaling with increasing number of cores for the same problem size

2. Make further performance improvements by speeding up individual subroutines in the hot path

In principle item 1 should be trivial as this application is embarrassingly parallel – "semantically" there are no dependencies between the colour computation for different pixels. We say semantically because, as we shall see, such dependencies can exist if the implementation uses certain constructs that can introduce contention where none should exist.

The most obvious obstacle to linear speedup is the fact that this application is in general very load-imbalanced. That can be mitigated to some extent by trying different scheduling clauses and varying the chunk size.

As for the algorithmic improvements to subroutines, `box::hit` is the most obvious candidate that we identified in section 2.3. Given its $\approx 20\%$ contribution to the baseline runtime, a 50% improvement would result in a reduction of the total runtime by 10%. We reasoned that anywhere between 30% to 50% speedup of the ray-box intersection should be achievable as the baseline imlementation calls the rect-hit functions on all six faces, whereas performant implementations in production ray-tracers use three faces or fewer (see for example Williams et al. 2005 and Majercik et al. 2018). That should give a roughly 50% reduction minus the overhead of culling faces that are not required.

Now that there is a decent understanding of the base code and how it works it is time to start analysing how to make it faster. The first step is to model the application. This was rather challenging due to the number of "zero-cost abstractions" and indirections in C++. The ray_color method in particular was hard to model because based on the material properties multiple separate paths will be taken, each with their own introduced complexity and abstractions. Due to this we opted for a more global model that describes the high level working of the application and various more detailed models for separate isolated algorithms which were selected for optimization. The model for the application is as described in eq. (1).

$$\begin{aligned}
T_{\text{raycast}} &= \max\{T_{\text{no hit}}, T_{\text{specular}}, T_{\text{hit}}\} \\
T_{\text{pixel}} &= \text{Samples} \times [0..\text{Max depth}] \times T_{\text{raycast}} \\
T_{\text{image}} &= W \times H \times (T_{\text{pixel}} + T_{\text{write pixel to shared buffer}}) \\
T_{\text{program}} &= T_{\text{initialization}} + T_{\text{image}} + T_{\text{write image}} + T_{\text{cleanup}}
\end{aligned} \tag{1}$$

The next step was to prepare the application to measure the run-times, write the output to a buffer instead of directly to stdout and finally to setup argument parsing for the CLI so that the configuration can be set outside of the program code. For the buffer we used `boost::multi_array` from the boost library[1]. This creates a multi-dimensional matrix backed by a 1 dimensional buffer. The output of the program is shown in fig. 2.

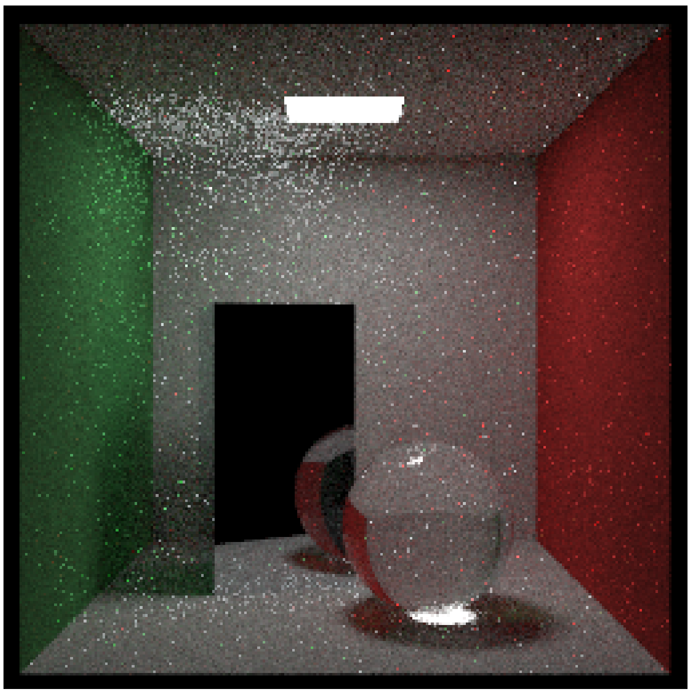


Figure 2: *Generated image for the test scene*

3.2 Parallelization

We simply parallelise the loop in line 2 in algorithm 1 by statically partitioning the work between a team of threads using OpenMP's *for* pragma. We expected to see a speedup with increasing number of threads, but with decreasing efficiency because of the load-imbalance [Treibig, Hager, and Wellein 2012].

3.2.1 Measured Performance and Analysis

However, we found instead a severe slowdown even with a small number of threads. Examples of execution times for this parallel version are shown in fig. 3.

A slowdown with increasing number of workers in a shared memory system is in most cases an indication of contention between threads due to e.g. synchronization constructs like mutexes, spin-locks, atomics et cetera [Treibig, Hager, and Wellein 2012].

So, we collected another runtime profile using perf, a sampling profiler. A quick glance down the right-most column in fig. 4[2] reveals that there are indeed many synchronization primitives being called – look for the following keywords. .

1. **futex:** a syscall for implementing locking
2. **raw_spin_lock:** self-explanatory
3. **try_to_wake_up:** Called when a thread is inactive due to wait for a shared resource to become available

[1] `https://www.boost.org/`
[2] This run is from one of our laptops and not the DAS as perf record was not working correctly over there.

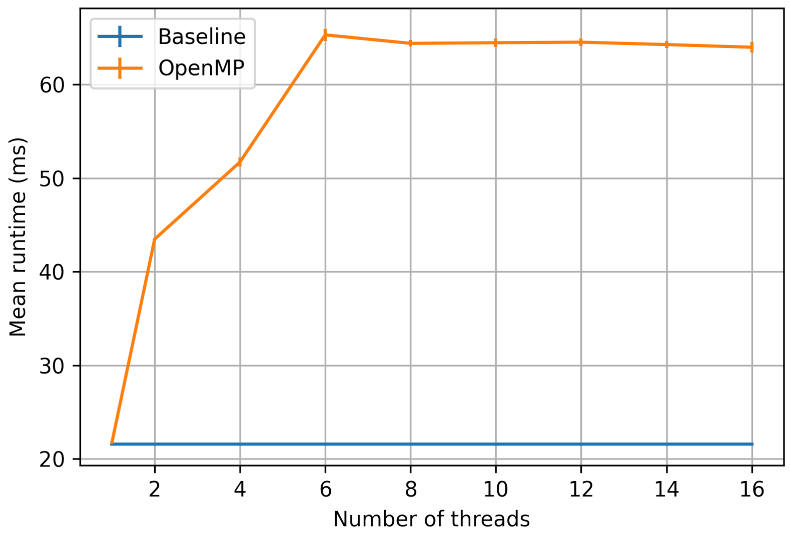
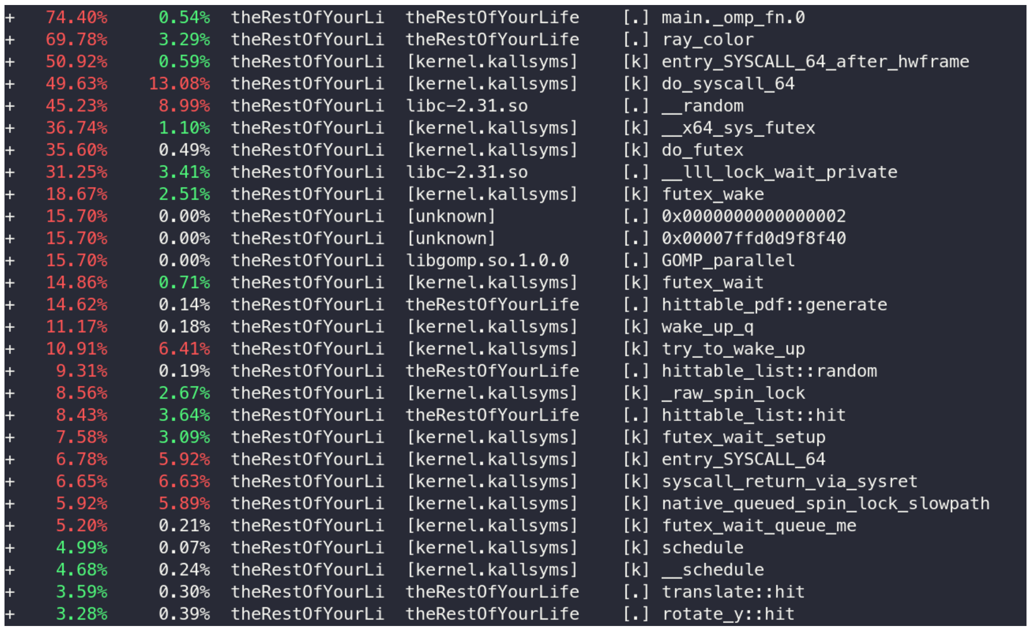


Figure 3: *Execution times of sample runs for the parallelised baseline version with different threads counts – smaller is better.*



```
+   74.40%     0.54%  theRestOfYourLi  theRestOfYourLife     [.] main._omp_fn.0
+   69.78%     3.29%  theRestOfYourLi  theRestOfYourLife     [.] ray_color
+   50.92%     0.59%  theRestOfYourLi  [kernel.kallsyms]     [k] entry_SYSCALL_64_after_hwframe
+   49.63%    13.08%  theRestOfYourLi  [kernel.kallsyms]     [k] do_syscall_64
+   45.23%     8.99%  theRestOfYourLi  libc-2.31.so          [.] __random
+   36.74%     1.10%  theRestOfYourLi  [kernel.kallsyms]     [k] __x64_sys_futex
+   35.60%     0.49%  theRestOfYourLi  [kernel.kallsyms]     [k] do_futex
+   31.25%     3.41%  theRestOfYourLi  libc-2.31.so          [.] __lll_lock_wait_private
+   18.67%     2.51%  theRestOfYourLi  [kernel.kallsyms]     [k] futex_wake
+   15.70%     0.00%  theRestOfYourLi  [unknown]             [.] 0x0000000000000002
+   15.70%     0.00%  theRestOfYourLi  [unknown]             [.] 0x00007ffd0d9f8f40
+   15.70%     0.00%  theRestOfYourLi  libgomp.so.1.0.0      [.] GOMP_parallel
+   14.86%     0.71%  theRestOfYourLi  [kernel.kallsyms]     [k] futex_wait
+   14.62%     0.14%  theRestOfYourLi  theRestOfYourLife     [.] hittable_pdf::generate
+   11.17%     0.18%  theRestOfYourLi  [kernel.kallsyms]     [k] wake_up_q
+   10.91%     6.41%  theRestOfYourLi  [kernel.kallsyms]     [k] try_to_wake_up
+    9.31%     0.19%  theRestOfYourLi  theRestOfYourLife     [.] hittable_list::random
+    8.56%     2.67%  theRestOfYourLi  [kernel.kallsyms]     [k] _raw_spin_lock
+    8.43%     3.64%  theRestOfYourLi  theRestOfYourLife     [.] hittable_list::hit
+    7.58%     3.09%  theRestOfYourLi  [kernel.kallsyms]     [k] futex_wait_setup
+    6.78%     5.92%  theRestOfYourLi  [kernel.kallsyms]     [k] entry_SYSCALL_64
+    6.65%     6.63%  theRestOfYourLi  [kernel.kallsyms]     [k] syscall_return_via_sysret
+    5.92%     5.89%  theRestOfYourLi  [kernel.kallsyms]     [k] native_queued_spin_lock_slowpath
+    5.20%     0.21%  theRestOfYourLi  [kernel.kallsyms]     [k] futex_wait_queue_me
+    4.99%     0.07%  theRestOfYourLi  [kernel.kallsyms]     [k] schedule
+    4.68%     0.24%  theRestOfYourLi  [kernel.kallsyms]     [k] __schedule
+    3.59%     0.30%  theRestOfYourLi  theRestOfYourLife     [.] translate::hit
+    3.28%     0.39%  theRestOfYourLi  theRestOfYourLife     [.] rotate_y::hit
```

Figure 4: *Runtime profile of baseline parallel application collected using perf record.*

We narrowed down to two prime suspects.

1. `rand`: a pseudorandom-number-generation (PRNG) function part of libc. This is a non-reentrant, thread safe function that uses a hidden state. Every call to `rand` mutates the internal state, and this write-access is most likely protected by synchronisation primitives.

2. `std::shared_ptr`: a smart pointer part of libc++11. This is an atomically reference counted (ARC) pointer to a heap-alloc'd object. Its reference count is updated whenever the pointer is copied or goes out of scope, i.e. whenever its constructors and destructors are called.

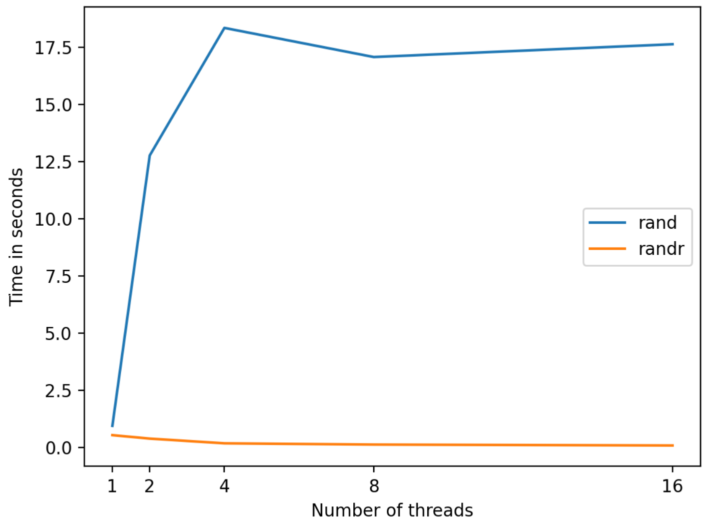


Figure 5: *Performance comparison of rand and rand_r on the DAS5*

3.3 Random-number-generation (RNG) in Multi-Threaded Contexts

3.3.1 Analysis

Consider the fifth line in the profile in fig. 4; it is something called `__random` from libc, which is called internally by `rand`. The baseline code uses `rand` for all its RNG; it is required for scattering off of diffuse materials, generating multiple rays per pixel, generating random vectors et cetera.

`rand`, on the other hand uses a single global state, and every call to it mutates this state. Therefore to make it MT-safe[3], access to this state is locked behind a lock. Consequently, every call to `rand` results in a critical section, and this is why it is such a significant part of the runtime profile.

A benchmark was used to measure the impact of replacing rand would have. This benchmark attempts to generate a very large number of random integers over various thread configurations. The results of these measurements are seen in fig. 5. From this figure it becomes clear as to why this prevented the previous version from achieving a linear speedup. A few other things also stand out in this graph. First, even with one thread rand_r performs slightly better then rand which can be attributed to not using any locking at all. Secondly, the increase in run-time between thread configurations for rand are quite significant. Even going from 1 to 2 threads increases the runtime by over 10 times.

3.3.2 Implementation

Ideally in a multi-threaded context each thread should have a thread-local RNG state that is independent of the states in other threads. The idiomatic way of doing this in C++ would be using `std::mt19337` instances from libc++, and instantiating one of each per thread, seeded by e.g. the thread IDs.

But, in the interest of minimizing refactoring effort we opted for `rand_r` from libc, which is the reentrant variant that can track multiple states independently. So, implementing this was a matter of using thread_local global integers for seeds and changing all `rand` calls to `rand_r`.

3.3.3 Measured performance

After this update we ran timings on this new version which resulted in the runtime graph as seen in fig. 6. In this graph it is shown that this is the first variant which actually gives us an

[3]Multi-thread safe, a POSIX safety attribute

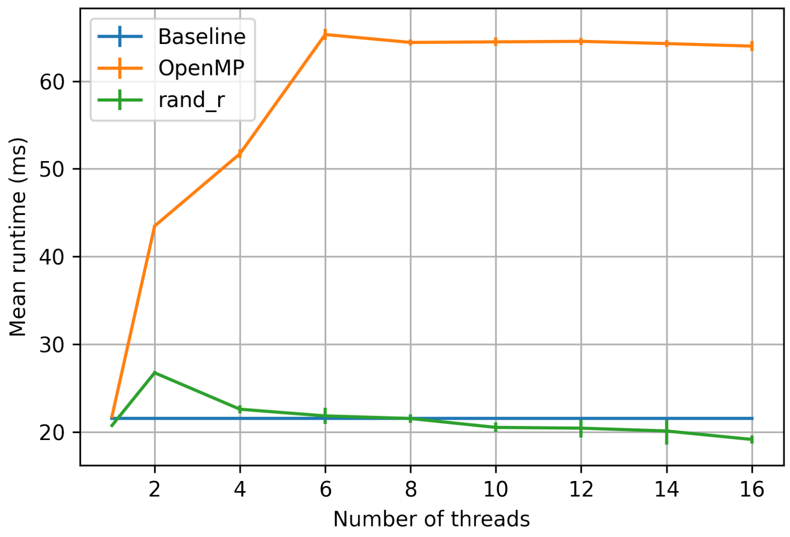


Figure 6: *Comparison of the different versions for various thread configurations*

performance improvement over the baseline version, even if multiple threads are required. It is still an odd profile and does not fit our expectations for this improvement which means that there is still another problem prevent us from gaining a predictable speedup.

3.4 `std::shared_ptr` is an anti-pattern

`std::shared_ptr` is an ARC smart pointer in libc++. It is used in the baseline code by geometry objects to refer to their materials, and by the scene to refer to its geometric objects.

3.4.1 Analysis and Implementation

A consequence of atomic refcounting is that an atomic increment/decrement is performed everytime any of the constructors/destructors of instances of this class are called. The baseline code uses these ARC pointers for all references to other classes. E.g. the world/scene object uses a contiguous array of shared pointers to the geometric objects. The geometric objects themselves contain shared pointers to their materials.

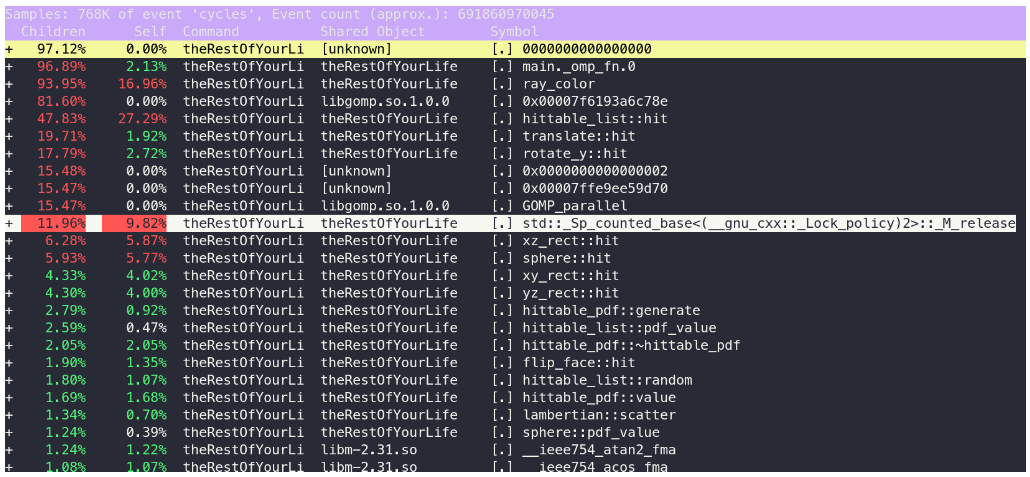
The ideal use case for shared pointers is when there are no clear lifetime bounds and/or ownership semantics. Both are clear in this case; the lifetime of the heap-allocated object is till just after the main loop, and the world owns its objects. All objects are accessed only through their member functions during the main iteration, and all member functions are const-qualified.

Therefore, we can simply get rid of the ARC pointers altogether as long as the heap-allocated objects are freed after the main loop. Two ways of doing this are raw pointers and `std::unique_ptr`s.

We chose to go for the raw pointers approach and changed all shared pointers to raw pointers. This completely got rid of any remaining synchronisation constructs from the main loop. Note that we could not completely remove the indirection by storing copies of the objects directly in the `std::vector` instead of pointers to them, because the objects are all different specialisations of a virtual class and use overloaded virtual functions. We are not aware of any polymorphic containers in the standard library and chose to stay close to the original implementation in this sense.

3.4.2 Shared pointer performance for multithreaded applications

A small benchmark was created to measure the performance of shared pointers over various thread configurations to estimate the performance gain from replacing them. This benchmark does a lot of concurrent loop iterations where the only operation performed is a copy of the shared pointer objects. If this is indeed the source of the witnessed slowdown then our expectation is that the



```
Samples: 768K of event 'cycles', Event count (approx.): 691860970045
  Children      Self  Command          Shared Object          Symbol
+   97.12%     0.00%  theRestOfYourLi  [unknown]              [.] 0000000000000000
+   96.89%     2.13%  theRestOfYourLi  theRestOfYourLife      [.] main._omp_fn.0
+   93.95%    16.96%  theRestOfYourLi  theRestOfYourLife      [.] ray_color
+   81.60%     0.00%  theRestOfYourLi  libgomp.so.1.0.0       [.] 0x00007f6193a6c78e
+   47.83%    27.29%  theRestOfYourLi  theRestOfYourLife      [.] hittable_list::hit
+   19.71%     1.92%  theRestOfYourLi  theRestOfYourLife      [.] translate::hit
+   17.79%     2.72%  theRestOfYourLi  theRestOfYourLife      [.] rotate_y::hit
+   15.48%     0.00%  theRestOfYourLi  [unknown]              [.] 0x0000000000000002
+   15.47%     0.00%  theRestOfYourLi  [unknown]              [.] 0x00007ffe9ee59d70
+   15.47%     0.00%  theRestOfYourLi  libgomp.so.1.0.0       [.] GOMP_parallel
+   11.96%     9.82%  theRestOfYourLi  theRestOfYourLife      [.] std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release
+    6.28%     5.87%  theRestOfYourLi  theRestOfYourLife      [.] xz_rect::hit
+    5.93%     5.77%  theRestOfYourLi  theRestOfYourLife      [.] sphere::hit
+    4.33%     4.02%  theRestOfYourLi  theRestOfYourLife      [.] xy_rect::hit
+    4.30%     4.00%  theRestOfYourLi  theRestOfYourLife      [.] yz_rect::hit
+    2.79%     0.92%  theRestOfYourLi  theRestOfYourLife      [.] hittable_pdf::generate
+    2.59%     0.47%  theRestOfYourLi  theRestOfYourLife      [.] hittable_list::pdf_value
+    2.05%     2.05%  theRestOfYourLi  theRestOfYourLife      [.] hittable_pdf::~hittable_pdf
+    1.90%     1.35%  theRestOfYourLi  theRestOfYourLife      [.] flip_face::hit
+    1.80%     1.07%  theRestOfYourLi  theRestOfYourLife      [.] hittable_list::random
+    1.69%     1.68%  theRestOfYourLi  theRestOfYourLife      [.] hittable_pdf::value
+    1.34%     0.70%  theRestOfYourLi  theRestOfYourLife      [.] lambertian::scatter
+    1.24%     0.39%  theRestOfYourLi  theRestOfYourLife      [.] sphere::pdf_value
+    1.24%     1.22%  theRestOfYourLi  libm-2.31.so           [.] __ieee754_atan2_fma
+    1.08%     1.07%  theRestOfYourLi  libm-2.31.so           [.] __ieee754_acos_fma
```

Figure 7: *Runtime profile of the parallelised code with all RNG changed to contention-free, reentrant variants*

time required to complete the look will increase with the number of threads due to the locking and atomic operations associated with the shared pointer. The results for this benchmark are as shown in fig. 8.
The results of this benchmark show that the runtime of this application increases dramatically when adding more threads.

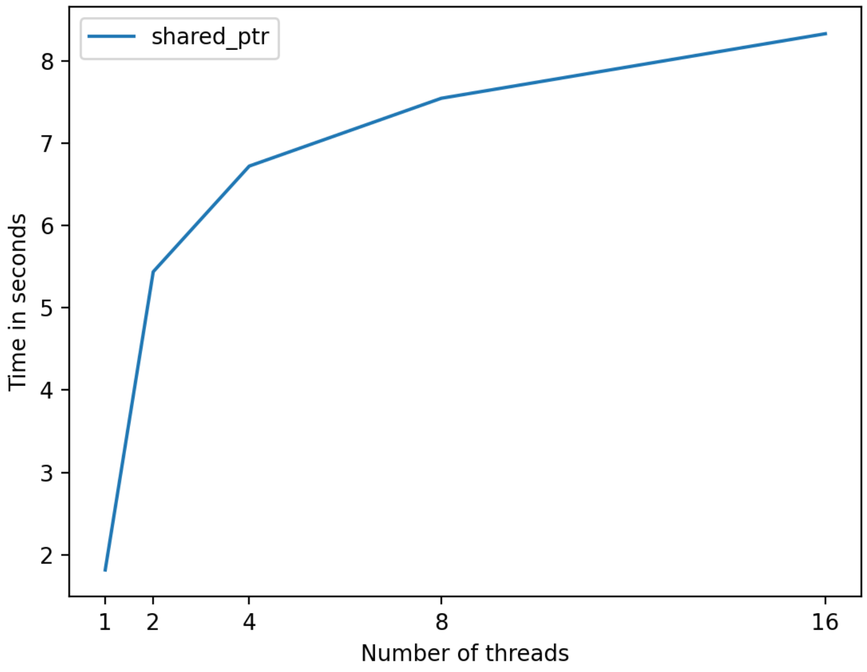


Figure 8: *Run-times for various thread configurations*

3.4.3 Measured performance

Figure 9 shows the increased performance after replacing the shared pointer objects with raw pointers. Do note that this is the first iteration of the project where each thread configuration is faster then the baseline, and that the performance scaling is as expected. Another important thing to note in this graph is that the new version of the application also runs faster on 1 node compared to the baseline version. This means that we also improved the performance compared to the baseline version without applying parallelism.

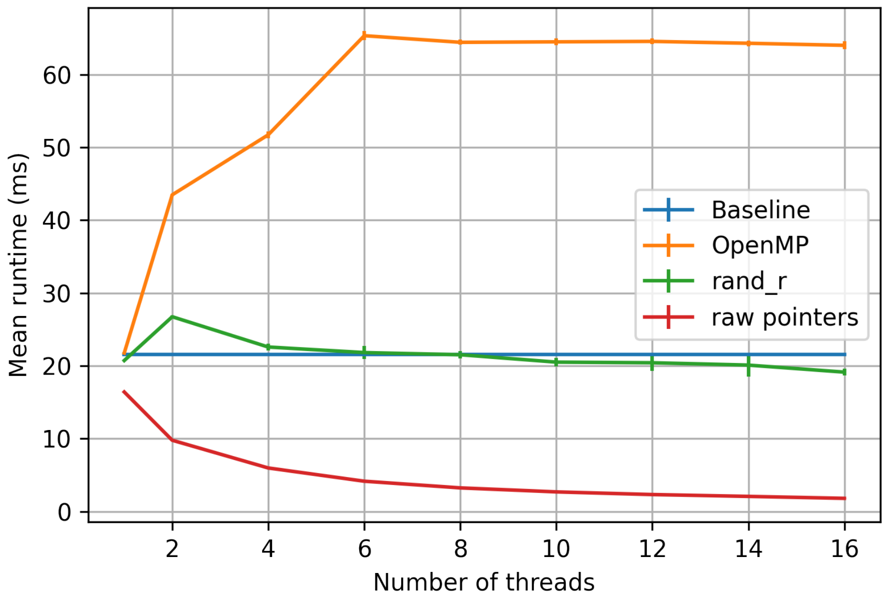


Figure 9: *Comparison of the different versions for various thread configurations*

3.5 Improved Ray-Box Intersection

After studying the gprof report fig. 1 we found that a lot of time is spend in the **_rect hit methods. A lot of these calls stem from the box hit method and as such if we'd manage to improve the performance of the box hit intersection should significantly limit the time spend in these specific methods.

3.5.1 Analysis

The original box intersection algorithm uses a list of six rectangles for the faces of the box. To then get the intersection point with the box it iterates of this list and performs a separate intersection call for each of the faces and returns the closest hit if any.
The model for this algorithm is as in eq. (2).

$$\begin{aligned} T_{\text{set face normal}} &= 5 \times \text{flop} \\ T_{\text{ray at}} &= 6 \times \text{flop} \\ T_{\text{test hit}} &= 6 \times \text{flop} \\ T_{\text{get hitpoint}} &= 6 \times \text{flop} + T_{\text{set face normal}} + T_{\text{ray at}} \\ T_{\text{box with ray hit}} &= 6 \times (T_{\text{function call}} + T_{\text{test hit}}) + 2 \times T_{\text{get hitpoint}} = 6 \cdot 6 + 2(6+5+6) = 36 + 34 = 70\text{flops} \\ T_{\text{box with ray miss}} &= 6(T_{\text{function call}} + \times T_{\text{test hit}}) = 6 * 6 = 36\text{flops} \end{aligned} \tag{2}$$

In this equation we assume that a ray always hits the farthest face first and the closer face second such that it does not prematurely exit out at the "test hit" method. From these numbers we find that a ray miss has about 51% as many floating point operations as the ray hit path.

3.5.2 Implementation

The algorithm used to improve the performance of the box hit algorithm is described on `https://iquilezles.org/articles/boxfunctions/`. This algorithm assumes that the box is perfectly

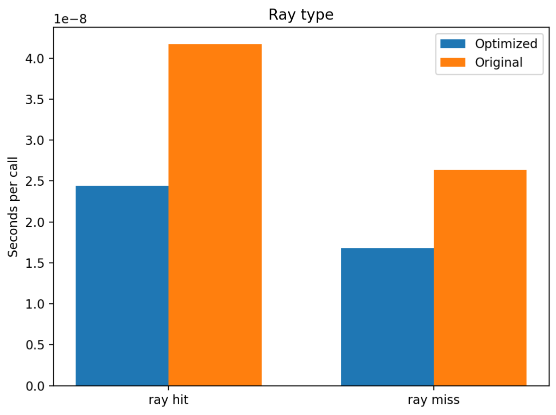


Figure 10: *Performance comparison between the original and optimized box hit algorithm*

centered in the world and that we have a vector with the radius per axis of the box. In our model a box can be anywhere in space, but it is aligned along the axises. This means that all we have to do to bring to box to the center is to translate the ray origin by the position of the box. The next step is to find the faces that the ray intersects first. This is done by deriving an vector s from the ray direction that has a value of 1 or -1 per axis depending of the ray direction on the same axes. Now the impact time values $T1$ and $T2$ can be calculated for every axis by the ray distance, ray direction, box radius and s vector. Finally the axis with the lowest T value is selected as the face that is intersected first by the ray, and the normal is then derived from this information.
The model describing this algorithm is as in eq. (3). By comparing these models you'd expect the miss operation to take more time due to more flops being performed and the ray hit to take less time because fewer flops are being performed.

$$\begin{aligned} T_{\text{set face normal}} &= 5 \times \text{flop} \\ T_{\text{ray at}} &= 6 \times \text{flop} \\ T_{\text{get hitpoint}} &= 6 \times \text{flop} + T_{\text{set face normal}} + T_{\text{ray at}} \\ T_{\text{box with ray miss}} &= 39\text{flops} \\ T_{\text{box with ray hit}} &= 39\text{flops} + T_{\text{get hitpoint}} = 39 + 17 = 56\text{flops} \end{aligned} \tag{3}$$

3.5.3 Comparison of the original and optimized algorithm

A benchmark was used to measure the performance improvement of the new algorithm. This benchmark constructs two identical boxes, one for the old algorithms, and one for the new one. Next is generates a few rays that hit the boxes, and a few that miss the boxes. Finally it repeats the ray intersection algorithm per ray type per box for a significantly large N to get the run-times of the algorithms. The results are shown in fig. 10.
The results do not entirely meet the expectations based on the models and the number of flops. For example, the 'ray miss' version of the new algorithm technically takes more flops, but it is still faster when measured. One cause for this is that the original algorithm performs six function calls regardless of hit or miss. The new algorithm has all of the logic present in one method only and thus saving on function calls.

3.5.4 Measured performance

Figure 11 shows the performance of the application after implementing the new algorithm. There was less of a speedup then hoped for based on these results, but there was a small performance improvement non-the less (almost a 10% performance improvement for single core). It also becomes clear that this improvement becomes negligible when many cores are used.

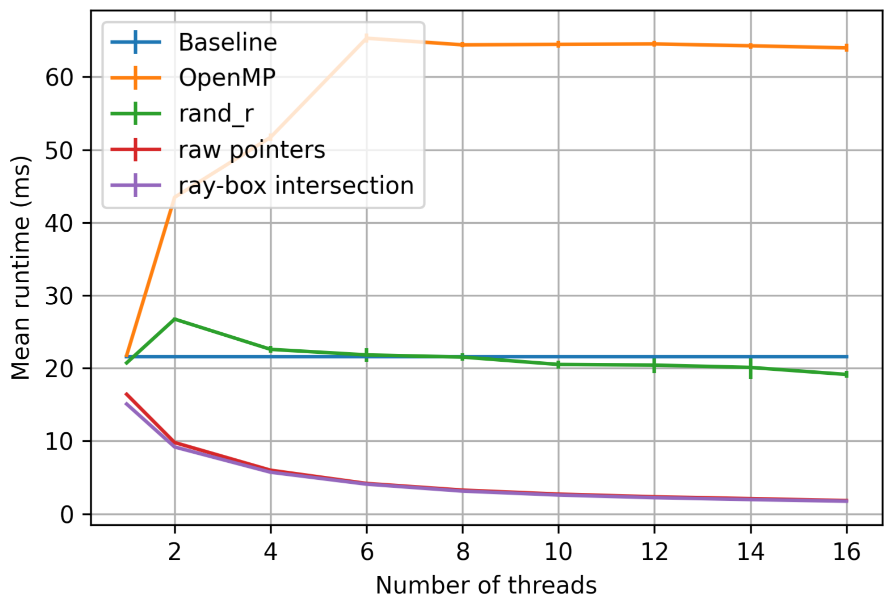


Figure 11: *Comparison of the different versions for various thread configurations*

4 Miscellaneous Optimisations

We tried making a few other changes that were not motivated by the profiler results. We present alternative possibilities for the `random_in_unit_sphere`, a subroutine that randomly generates a vector inside the unit sphere (distributed uniformly) and discuss the challenges in modeling them. This is used by diffuse materials for scattering light rays in random orientations.

We found the model construction and the verification process very interesting for multiple reasons – it offered great insight into trade-offs between a stochastic algorithm and deterministic algorithms, and those between different floating point arithmetic functions. In addition to this, verifying and validating the model led us to discover subtleties involved in microbenchmarking - subtleties whose causes lay deep in the microprocessor design.

4.1 Baseline `random_in_unit_sphere`

The baseline version employs an accept-reject method by generating a vector point uniformly in the bounding box of the unit sphere, and then checking if that vector also lies in the unit sphere. If yes, then this vector is returned. Else it keeps trying until it gets it.

Algorithm 2 Baseline implementation of `random_in_unit_sphere`

```
while true do
    $x \sim \text{Uniform}(-1, 1)$
    $y \sim \text{Uniform}(-1, 1)$
    $z \sim \text{Uniform}(-1, 1)$
    $l \leftarrow x^2 + y^2 + z^2$
    $v \leftarrow \texttt{vec3}(x, y, z)$
    if $l > 1$ then
        continue
    end if
    ret $v$
end while
```

We can construct a simple probabilistic model for this subroutine as the distributions are stationary.

Analytical Model

We make the following assumptions in this model.

1. The uniform RNG is follows ideal uniform distribution.
2. Speculative execution does not take place.
3. There is no instruction-level parallelism (ILP).

The ratio of the volume of the unit sphere and that of its bounding box is

$$p = \frac{\frac{4}{3}\pi}{2^3} = \frac{\pi}{6} \approx 0.52 \tag{4}$$

The loop body is essentially a Bernoulli trial with probability of success given by p in (4). The number of times the loop in line 1 iterates is thus a geometric random variable, i.e. the number of trials that it takes to get the first successful trial.

Let this random variable be denoted by L. Then, we have the expectation value and variance of L given by

$$\text{Exp}[L] = \frac{1}{p} = \frac{6}{\pi} \approx 1.9 \tag{5}$$

$$\text{Var}[L] = \frac{1-p}{p^2}. \tag{6}$$

Therefore, we expect from (5) that on an average the loop in line 1 will perform roughly two iterations. Let T_{i} be the execution time per iteration of the loop. Then, the total execution time T_1 of algorithm 2 is the random variable

$$T_1 = L \cdot T_i. \tag{7}$$

The expectation value of the runtime along with its variance is therefore given by

$$\text{Exp}[T_1] = \frac{1}{p}T_i = \frac{6T_i}{\pi} \tag{8}$$

$$\text{Var}[T_1] = T_i^2\frac{1-p}{p^2}. \tag{9}$$

T_i can be estimated by adding the latencies/execution times of the major operations, i.e.

$$T_i = 3T_{\text{rand}} + T_{\text{normsq}} + T_{\text{vec3}} + T_{\text{cond}} \tag{10}$$

Where $T_{\text{rand}}, T_{\text{normsq}}, T_{\text{vec3}},$ and T_{cond} are the execution times of the uniform RNG, calculating the square of the L2 norm (l in algorithm 2), the `vec3` constructor, and the conditional respectively. So, validating this model requires microbenchmarking the individual functions as well as the subroutines as a while.

We would also like to validate the higher level probabilistic model summarised by (8) and (9).

4.2 Loop-less Version: Spherical Coordinates

We can get rid of the infinite loop by creating a uniform distribution directly in the unit sphere instead of first generating in the bounding box. The trade-off here is between unnecessary loop iterations and some extra flops. In this section and the next one we show examples of two such algorithms, propose models for them, and document our calibration attempts.

Algorithm 3 Loop-less implementation of `random_in_unit_sphere`; spherical coordinates

```
1: $x \sim \text{Uniform}(0, 1)$
2: $r \leftarrow x^{\frac{1}{3}}$
3: $\theta \sim \text{Uniform}(0, \pi)$
4: $\phi \sim \text{Uniform}(0, 2\pi)$
5: $z \leftarrow r \times \cos\theta$
6: $r_s \leftarrow r \times \sin\theta$
7: $x \leftarrow r_s \times \cos\phi$
8: $y \leftarrow r_s \times \sin\phi$
9: ret $\texttt{vec3}(x, y, z)$
```

Analytical Model

Under the same assumptions as before, we can construct the model the execution time as

$$T_2 = 3T_{\text{rand}} + T_{\text{cbrt}} + 2T_{\text{sin}} + 2T_{\text{cos}} + 4T_{\text{mul}} + T_{\text{vec3}}, \tag{11}$$

where $T_{\text{sin/cos}}$ and T_{mul} are the execution times of the `sin/cos` functions from libm, and T_{mul} is that for floating point multiplication.

We anticipated that the flops for conversion from spherical to Cartesian coordinates would be very high. There is also the possibility of generating directly in the Cartesian coordinates.

4.3 Loop-less version: Cartesian Coordinates

Algorithm 4 Loop-less implementation of `random_in_unit_sphere`; Cartesian coordinates

```
1: $x \sim \text{Uniform}(-1, 1)$
2: $y_l^2 \leftarrow 1 - x \cdot x$ 4
3: $y_l \leftarrow \sqrt{y_l^2}$
4: $y \sim \text{Uniform}(-y_l, y_l)$
5: $z_l^2 \leftarrow y_l^2 - y \cdot y$
6: $z_l \leftarrow \sqrt{z_l^2}$
7: $z \sim \text{Uniform}(-z_l, z_l)$
8: $v \leftarrow \texttt{vec3}(x, y, z)$
9: ret $v$
```

Analytical Model

See the pseudocode in algorithm 4. So, we can in the same way model the execution time for this kernel as

$$T_3 = 3T_{\text{rand}} + 2T_{\text{sqrt}} + 2T_{\text{mul}} + 2T_{\text{sub}} + T_{\text{vec3}} \tag{12}$$

4.4 Validating the Models

We followed the following steps to validate the models of section 4.1, section 4.2, and section 4.3.

1. Benchmark the individual subroutines – baseline, spherical and Cartesian.
2. Microbenchmark the individual functions
   (a) Square root, cube root ($T_{\text{sqrt}}, T_{\text{cbrt}}$)
   (b) Floating point addition, subtraction, and multiplication ($T_{\text{add}}, T_{\text{sub}}, T_{\text{mul}}$)
   (c) Uniform RNG functions (T_{rand})
   (d) Trigonometric functions ($T_{\text{sin}}, T_{\text{cos}}$)

(e) The `vec3` constructor (T_{vec3})

3. Use values obtained from item 2 in the model and compare against values obtained from item 1.

4.5 Adventures in Mircobenchmarking

We would now like to use our model to better understand why this performance difference is observed. This is where things start becoming non-trivial for us. Two questions need to be addressed here at every point in this process: is it possible to accurately benchmark the individual functions listed in section 4.4? How do we verify the benchmarks? Do the timing measurements make sense?

We used the open source C++ library Google benchmark [5] to benchmark individual instructions and functions. The code to be benchmarked is iterated in a range-based for loop over a state object that tracks the timing measurements.

```
static void BM_Func(benchmark::State& state) {
    for (auto _ : state) {
        /* code to be benchmarked */
    }
}
```

The nature of the loop turns out to play an important role in collecting accurate timing measurements.[6]

Benchmarking Individual Components

Consider an example of benchmarking one of the components – the square root function `sqrt` from libc (which eventually compiles down to one of X86-64's `sqrt` instructions) . The first version of our benchmark code looked like so.

```
static void BM_Sqrt(benchmark::State& state) {
    double x = random_double(), y = 0.01;
    for (auto _ : state) {
        y = sqrt(x);
        escape(&y); }}
```

Here, `escape` is an inline assembly function that does not compile to anything, but prevents the compiler from performing dead-code elimination.[7] We verified this by inspecting the inline assembly.

```
static void escape(void *p)
    asm volatile("" : : "g"(p) : "memory");
```

This turns out to be very problematic as seen in fig. 12 – apparently the benchmark loop with `sqrt` in there is equivalent to a `noop` loop. Reference manuals place the latency of the x86-64 `sqrt` instructions in the interval > 10 and < 30 clock cycles (see for example Fog 2022). Therefore, we conclude that our benchmark is not measuring what we want to measure (i.e. the full latency of the function).

Our hypothesis is that this occurs due to ILP in the microprocessor as there are no loop-carried dependencies, and the results of each iteration's computation are not being used anywhere. Consequently, the processor can take full advantage of multiple-issue and pipelining. Intel's microarchitectures also have loop-stream detectors (LSDs) that can detect the presence of a loop in the stream of instructions and eliminate the loop overhead upon detection. The LSD could also be contributing to the efficiency of ILP. We propose to change the benchmark loop to a *fixed-point iteration (FPI)*.

```
static void BM_SqrtFPI(benchmark::State& state) {
    double x = random_double();
    for (auto _ : state) {
```

[5] `https://github.com/google/benchmark`

[6] The functions that we microbenchmark are very tiny and using manual timers was resulting in very slow convergence. The benchmark library is able to get around this.

[7] See `https://youtu.be/nXaxk27zwlk?t=2476` for details.

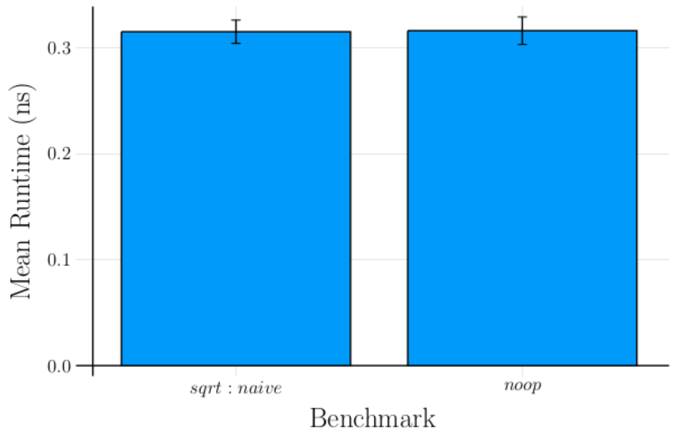


Figure 12: *Timing result of benchmarking – naive approach.*

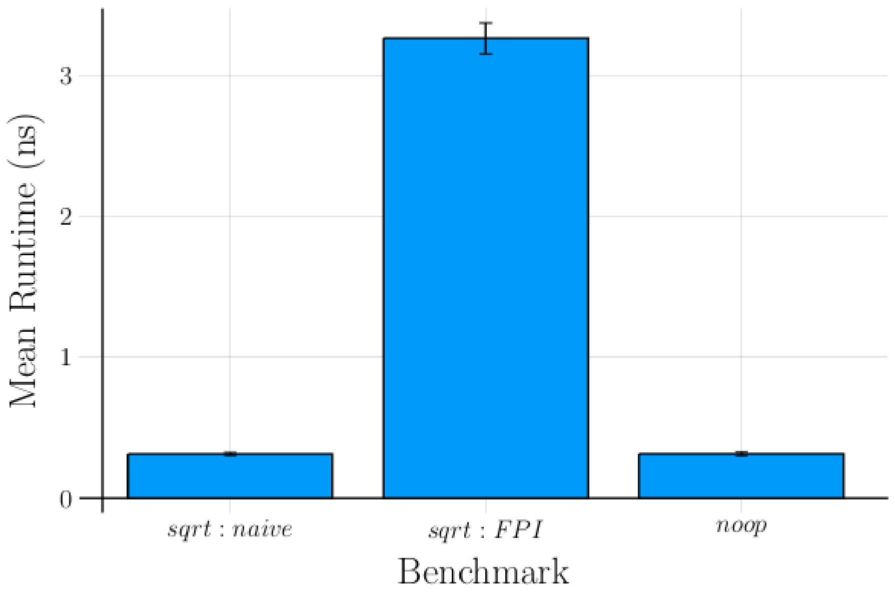


Figure 13: *Benchmark results for square root with the fixed-point iteration method*

```
        x = sqrt(x); }
    escape(&x); }
```

This introduces a loop-carried dependency and completely eliminates any possibility of ILP. The benchmark results are shown in fig. 13. This FPI method can be used to benchmark any function whose range is a subset of its domain. However, this benchmark might still be problematic if the latency of the **sqrt** instruction depends on the argument, because the fixed point iteration $x_{n+1} = f(x_n)$ converges a the solution of the equation $x = f(x)$, and the convergence is fairly rapid on the “timescale” of the benchmark loop. This holds for $f(x) \in \{\sin x, \cos x, \sqrt{x}\}$. The results for the remaining components are shown in table 1.

Finally, we have the benchmarks for the complete subroutines themselves in fig. 15 and table 2. The cartesian version appears to be a lot faster than the baseline implementation, but that conclusion can only be drawn if our benchmarks are representative of the actual performance properties in production runs. We shall touch upon this in subsequent sections.

Verifying the Probabilistic Model

We also wanted to verify the simple probabilistic model that we developed for the baseline accept-reject method in section 4.1 – we want to start by verifying if (8) and (9) hold or not. This

Subroutine/Instruction	Mean Runtime (ns)	Standard Deviation (ns)
`sqrt`	3.26	0.11
`sin`	15.8	0.011
`cos`	17.0	0.5
`random_double`	5.68	0.19
`cbrt`	34.9	0.03
`vec3` ctor	0.630	0.02
`vec3::length_squared` + `vec::random`	16.2	0.088
`vec3::random`	16.3	0.052
Double Precision Multiplication	3.47	0.12
Double Precision Addition	2.83	0.015
noop	0.316	0.013

***Table 1:** Benchmark results – individual components of the random in sphere subroutines*

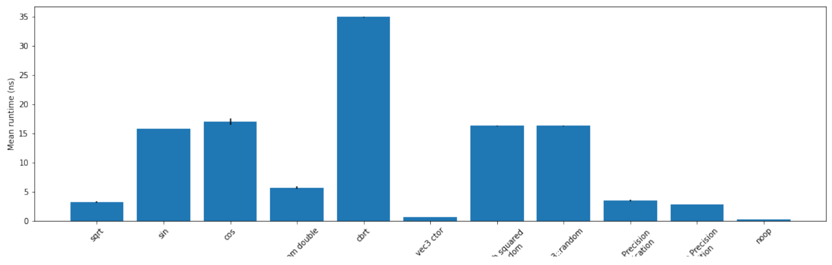


***Figure 14:** Benchmark results of component instructions/subroutines.*

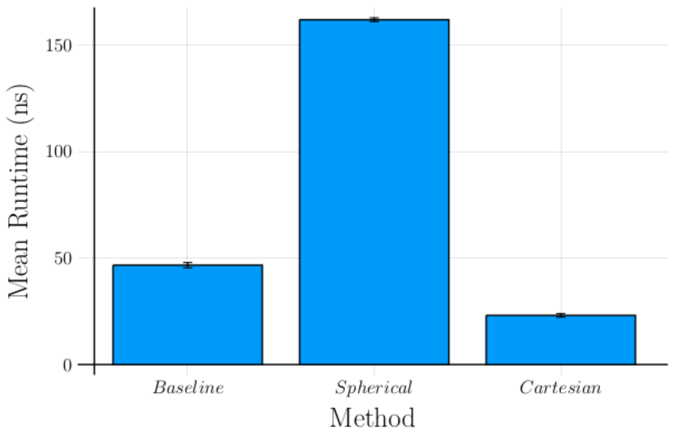


***Figure 15:** Benchmark runtimes of the different random in unit sphere methods.*

Method Benchmarked	Mean Runtime (ns)	Standard Deviation (ns)
Baseline	46.7	1.2
Spherical	162.0	0.9
Cartesian	23.4	1.7
Baseline – single iteration	24.6	0.1

***Table 2:** Benchmark results, mean and standard deviation, of the random in unit sphere methods (with single loop iteration separately in the last row)*

Method	Mean Runtime - analytical (ns)	Mean Runtime - measured (ns)	% Error
Baseline	46.7	34.0	27.2
Spherical	162.0	132.5	18.2
Cartesian	23.4	36.8	57.3

Table 3: *Analytical Predictions vs Measured Values*

Method	Branches	Branch Misses	Branch Miss Rate
Baseline	21,434,632,853	959,886,482	4.48%
Spherical	35,884,239,748	1,284,655,783	3.58%
Cartesian	482,571,238	482,572	0.00%
Baseline – single iteration	25,609,565,692	1,056,532,476	4.13%

Table 4: *Estimates of total number of branch misses and branch-miss rates for the individual methods using performance counter data. Google benchmark runs the the benchmark loop a variable number of times – loop termination depends on achieved standard deviation. The metric of interest is hence the branch-miss rate.*

requires us to benchmark just the loop body in algorithm 2 to estimate T_i, benchmark the whole random in unit sphere subroutine to estimate T_1, and then observe the mean and variance of their ratio – the mean should be approximately equal to $p = \pi/6$. We see from table 2 that $T_i/\mathrm{E}[T_1] \approx 24.6/46.7 \approx 0.527$, and the analytically anticipated value is $\pi/6 \approx 0.523$, resulting in a relative error of 0.7%. We deem this acceptable.

However, things do not quite work out when we verify equations (10), (11), and (12). The relative errors between our models and the benchmark measurements are summarized in table 3.

We also decided to run the benchmarks individually under perf stat. On collecting branches and branch-miss performance counter aggregates for the benchmarks individually, we found that the baseline and spherical implementations were experiencing a disproportionately large rate of branch mispredictions (see table 4). It is immediately obvious that the branch misses are coming from the conditional in the loop body in algorithm 2 because (8) holds and we can see in table 4 that both the baseline implementation and its loop body in isolation experience a lot of branch misses.

It might not be immediately apparent why algorithm 3 has so many branches in the first place. The trigonometry functions in libm (in our case `sin` and `cos`) use conditionals internally; they are implemented as interpolating polynomials in $[0, \frac{\pi}{2}]$, and use the periodicity of the functions to translate arguments arguments outside this interval into this interval. Conditionals are used to select the final branch-free interpolating polynomial microkernel in this manner.

In this case, it is in general possible that while these subroutines actually experience a different rate of branch misses in the actual raytracer. Therefore, our benchmarks for the baseline and spherical implementations are not representative.

In summary, we have the following main issues [8].

1. Our model assumes no speculative execution, which can no longer be considered a valid assumption.
2. Incorporating speculative execution into the model requires information about
   (a) branch misprediction penalty
   (b) successful prediction bonus
   (c) branch-misprediction rate in the raytracer.
3. Given that ILP made non-trivial difference in our benchmarks, the model assumption of no ILP now stands on shaky ground and should be done away with.

One method typically used to minimise branch misses on individual branch instructions is to use compiler pragmas that enable to programmer to inject knowledge about which branch is

[8] in addition to the minor issues like possibly argument-dependent instruction latencies

expected to occur more frequently than the alternative.[9] This does not work in this case because the conditional only has a consequent and no alternative – changing the likelihood simply flips the condition flags of the conditional jump instruction.

5 Conclusion

We were successfully able to perform profiler-guided optimisation on this ray-tracer to some extent. We removed all sources of contention by changing RNG to MT-safe, reentrant variants, and got rid of atomic reference counting by reasoning about ownership semantics, lifetime bounds, and const-correctness of the heap-alloc'd objects and accordingly changing shared pointers to raw pointers. We also managed to significantly accelerate the ray-box intersection algorithm by changing it to a more sophisticated algorithm.

Our modeling approach for individual kernels for the `random_in_unit_sphere` appears at first glance to be partially successful; modeling the outer loop's conditional in the baseline as a geometric random variable results in an accurate estimate of the average number of loop iterations per call. But, the finer grained model that relied on adding up the latencies of individual functions and instructions did not fare well as we realised from the benchmarks and performance counter aggregates that an accurate model would have to take into account ILP and speculative execution.

6 Future Work

We believe that future work should focus on developing better performance models for individual microkernels in the application – the raytracer is a essentially a collection of hundreds of tiny kernels that compose together to develop individual pixels. Such performance models would allow for better insight into remaining inefficiencies in the kernels.

Getting better insight into why some kernels perform better than others would require developing more accurate models that take into account ILP and speculative execution. A possible direction of work on this could be studying the target microarchitecture more rigourously (e.g. using Agner Fog's manuals, or other recommended resources), comparing against tools like OSACA or llvm-mca, and using fine-grained performance counter data. A statistical modeling approach could be used for incorporating branch mispredictions into the model. As a starting point we can have a simple statistical model with a matrix of tunable parameters that includes the misprediction penalties and correct prediction bonuses for all branches in a conditional. The model input would then be the branch-miss rate, which can be measured using performance counters, and the target would be a measure of the execution time.

We also saw that the trigonometric functions in libm are inefficient as they assume that the argument can be anywhere on the real line. But the angles generated throughout the entire program are guaranteed to be in $[0, 2\pi]$, and in some cases even smaller intervals. So it is possible to create more optimised trigonometric functions that assume that the argument belongs to this range only and eliminate the associated overhead. This would require some study in numerical mathematics as well as we would have to prove error bounds for these new functions. Or we can also use inline assembly or compiler builtin functions to directly call the X86 trigonometric instructions. We can also use the abovementioned speculative execution and ILP based model to make performance predictions for the new trigonometric kernels.

Once models are successfully constructed, the same approach can be used to get better insight into the multifarious microkernels utilised in the raytracer in order to develop a better understanding of the application as a whole.

References

Fog, Agner (2022). "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers". In: *Copenhagen University College of Engineering.*

[9]See for example GCC's `__builtin_expect` at `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

Majercik, Alexander et al. (2018). “A ray-box intersection algorithm and efficient dynamic voxel rendering”. In: *Journal of Computer Graphics Techniques Vol* 7.3, pp. 66–81.

Treibig, Jan, Georg Hager, and Gerhard Wellein (2012). “Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering”. In: *European Conference on Parallel Processing*. Springer, pp. 451–460.

Williams, Amy et al. (2005). “An efficient and robust ray-box intersection algorithm”. In: *ACM SIGGRAPH 2005 Courses*, 9–es.