SPHERE - AN OPTIMIZED SPHERE TRACER IN C++

André Gaillard, Christopher Rafft, Jens Eirik Saethre, David Spielmann

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Sphere tracing is a technique to realistically render implicit surfaces closely related to ray tracing and ray marching. In this work, we derive a baseline implementation of a sphere tracer and apply successive, thematic optimizations which include but are not limited to vectorization, strength reductions, as well as multi-threading, and are handcrafted for an Intel processor of the Skylake microarchitecutre. In doing so, we achieve a significant increase in performance and a speedup of 327, enabling us to render a complex image in Full HD resolution in a matter of seconds.

1. INTRODUCTION

Creating photo-realistic depictions of complex sceneries has been of interest to humankind for centuries. In the Renaissance, it were paintings like Raphael's *School of Athens* or Da Vinci's *The Last Supper* that drew attention due to their accurate perspective projections, a novelty at the time. In contrast, today such realistic graphics are mostly computer-generated, and they dominate the world we live in: They appear in computer games [1], movies [2], real estate marketing [3], scientific visualisations [4], modern art, and many other fields. Traditionally, real-time applications relied on algorithms such as z-buffering [5], whereas static images were produced via more realistic techniques such as ray tracing [6, 7] or ray marching [8]. However, with the increase in processor clock frequencies and the advent of general purpose GPUs, the latter techniques have recently become attractive to real-time applications such as gaming [9].

The idea behind these techniques is to trace rays from the camera lens through the image plane (consisting of pixels) onto the scene, and let them be reflected or absorbed by objects they hit. While ray tracing is the more prominent technique and is e.g. used in new computer games such as *Battlefield 5* and *Cyberpunk 2077* [9], it is computationally expensive. This is due to the fact that intersections between rays and objects have to be computed directly, which translates to solving complex equations. Ray marching, on the other hand, uses an iterative approach. It relies on computing much cheaper distance functions that return the distance of the current position on the ray to an object in the scene.

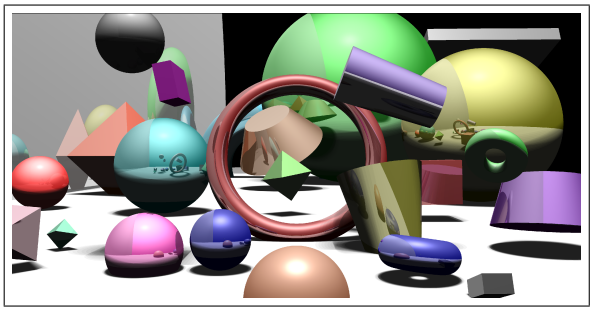


Fig. 1: Example scene. The image was produced using SPHERE and contains 24 visible and 72 invisible objects.

In this work, we consider *sphere tracing*, a special form of ray marching used to render implicit surfaces that was introduced by Hart in [8]. This technique uses a variable step size that is adapted in each step rather than having a fixed one. The step size is determined in a manner that maximizes the distance marched forward in each step while guaranteeing not to intersect any object.

Our Contribution. We present SPHERE [10], a rendering system that is based upon sphere tracing. It can be used to produce photo-realistic images like the one shown in figure 1 from a scene description. SPHERE is highly optimized for processors of Intel's *Skylake* microarchitecture, and exploits both their AVX2 and multi-core properties.

Related Work. Over the last decades, various rendering algorithms have been proposed. While the idea behind ray tracing was first introduced in 1525 by Albrecht Dürer in [11], it was Whitted who first used this algorithm for digital rendering in [12]. This algorithm was refined in [6] by averaging over multiple rays to allow for phenomena such as motion blur or softer shadows. In [13], the authors use Monte Carlo methods to randomly sample light rays in the scene to produce realistic renderings, an approach they call the *Metropolis Light Transport*. Ray marching methods, on the other hand, were first introduced by Hart in his work on sphere tracing [8], which constitutes the foundation of our work. Recent work by Seyb et al. extends this method to non-linear sphere tracing that allows to render deformed signed distance fields [14]. They do so by efficiently integrating ordinary, non-linear differential equations.

2. BACKGROUND

The main goal of this section is to introduce the concept of sphere tracing as a tool to render scenes in computer graphics. Towards that goal, we first explain the idea behind ray tracing and ray marching. We then discuss the sphere tracing algorithm and analyze its asymptotic complexity. Finally, we define the cost measure that is used throughout the remainder of this work.

2.1. Ray Tracing and Ray Marching

Both ray tracing and ray marching are based on the same idea. Consider a scene with a light source, a camera, an image plane (consisting of pixels), and a few objects. We shoot rays from the camera lens through each pixel in the image plane. When a ray hits an object, a new ray is cast to check if the point hit is illuminated by a light source. If that is the case, we calculate the reflection direction and use this direction to cast a reflection ray. This is repeated recursively, resulting in a sequence of rays. Every object that is hit by a subsequent reflection ray contributes to the shading of the original hit depending on the material parameters, shadows, and illumination. To avoid infinite recursions, the ray shooting is aborted if any of the following three scenarios occurs: a ray hits an object that has no reflective material property, a fixed threshold for the number of reflections is exceeded, or a fixed threshold for the total ray length is exceeded [15].

What remains to be discussed is how these methods determine the points where rays and objects intersect, and it is here where the two methods differ. Traditional ray tracing explicitly calculates the intersection of a ray with an object, which becomes highly non-trivial and expensive for complex object types. Ray marching, on the other hand, relies on an iterative approach and is mainly used to render implicit surfaces. These are surfaces that can be described via implicit functions, e.g. $F(x, y, z) = x^2 + y^2 + z^2 - r^2$ for a sphere at the origin with radius r. These functions are considered implicit because they are not constructive in the sense that they cannot be used to build the object's surface. Instead, for any point $\mathbf{x} \in \mathbb{R}^3$ they indicate the squared *signed* distance of $\mathbf{x}$ to the surface of the object. Hence, these functions are also called *signed distance functions* (SDFs) and they are often a lot cheaper to compute than the equations in ray tracing.

We can use SDFs for ray marching as follows. We define a step size and use it to march on the ray. In each step, the signed distances to all objects are calculated. The ray intersects an object if the sign of the distance changes, i.e. if it goes from $d > 0$ to $d < 0$. The intersection point can then be found with binary search within the last step. However, traditional ray marching suffers from one major issue that is related to the choice of the fixed step size. Figure 2 illustrates this problem.

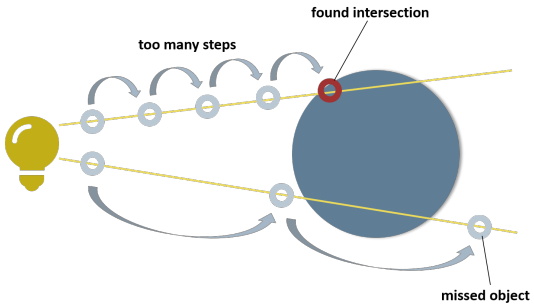


Fig. 2: Issue with traditional ray marching. A too small step size leads to many unnecessary steps and thus slows down the algorithm, while a too large step size can result in objects being missed.

2.2. Sphere Tracing

Sphere tracing alleviates this issue by making the step size variable and maximizing it in each step. It does so in three phases:

(i) Compute the signed distances from the current position on the ray to all objects.

(ii) Take the minimum of all signed distances $d_{\min}$ and draw a virtual sphere with radius $d_{\min}$ around the current point on the ray.

(iii) March forward with the new step size set to $d_{\min}$.

The sphere drawn in (ii) corresponds to the *safe zone*. From the current position on the ray one can move to any point within this sphere and is guaranteed not to intersect any object. This safe zone is maximal in the sense that the above property would be violated for any larger sphere. Note that this sphere is not actually computed but only serves to illustrate the idea behind sphere tracing. By stepping as described in (iii), one can never transgress an intersection point. Once the distance to an object's surface is below a very small threshold, the intersection point is found and the algorithm is called recursively as discussed in section 2.1.

Asymptotic Complexity. Rendering an $m \times n$ pixel image that contains k objects using the sphere tracing algorithm has an asymptotic complexity of $\mathcal{O}(kmn)$. The reason for this is that for every pixel the distance to each object has to be computed at least once, but at most a constant number of times. This is due to the fixed tracing termination thresholds mentioned in section 2.1. Moreover, the SDFs can be computed in constant time. Note, however, that the constants hidden in this notation can be incredibly large.

2.3. Cost Measure

The exact cost in terms of floating point operations of the sphere tracing algorithm depends not only on the image size

and the size of the scene, but also on its contents, e.g. where certain objects are placed and the material properties of the objects (e.g. reflectivity). For this reason, it is not possible to give a general mathematical expression that captures the cost. Instead, we implement a flop-counting functionality in our library that is activated in a special build mode and provides exact flop counts. For a set of floating point operations S it uses the following cost measure:

$$C(m,n,k) := \sum_{s \in S} C_s \cdot N_s, \tag{1}$$

where N_s denotes the number of operations of type s. For the costs, we define $C_s := 1$ for basic arithmetic operations, $C_s := 1$ for logical operations like AND, OR, AND NOT, and $C_s := 30$ for complex math functions like `tanh()`, `exp()`, `pow()`, or `sqrt()`.

3. OPTIMIZATION ROADMAP

In this section we present the different versions of our algorithm. First, we provide a brief overview of the baseline version. Taking on from there, we discuss the successive, thematic optimizations that we applied and which are outlined in figure 3. Finally, we briefly touch upon other optimizations and considerations not included in any of our algorithm's versions.

3.1. Baseline

The baseline consists of a straightforward implementation of the sphere tracing algorithm inspired by [16], distance functions for different types of shapes as given in [17], shading and shadow features as well as the necessary infrastructure to load a scene and render the output of the algorithm to an image file. It is capable of rendering a scene containing an arbitrary number of six pre-defined shape types illuminated by a single light source. The shapes can be rotated and positioned willfully. We incorporate Phong shading effects [18] that provide ambient and diffuse lighting as well as specular highlights. Furthermore, we support reflections up to a certain distance. These effects are controlled by parameters predefined in the input scene description. To make the scene more realistic, we let the objects cast shadows with configurable complexity in terms of diffusion. As detailed shadows heavily increase the runtime, we restrict ourselves to simple ones in our experiments.

SPHERE is implemented in an object-oriented programming (OOP) manner: Entities such as the scene, the rendering engine, and all the shapes are C++ classes implemented in individual source files, the shape classes all inheriting from a common base class `Shape`. Moreover, we use concepts such as operator overloading on a custom `Vector` class. Needless to say, this version's focus lies on correctness and being extendable rather than being performant.

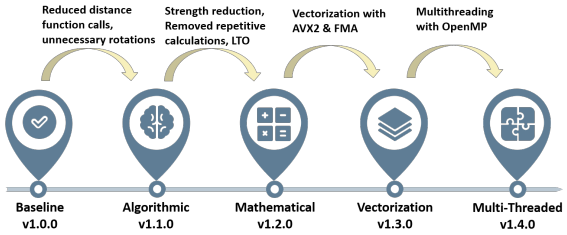


Fig. 3: Optimization roadmap. We successively apply thematically grouped optimizations from the baseline to the final, multi-threaded version.

3.2. Algorithmic Optimizations

Our first optimizations are of algorithmic nature. Profiling the baseline with the Intel VTune profiler[1] unveils that most of the runtime is spent in the distance functions. Consequently, the first objective is to reduce the number of calls to them by using a smarter algorithm. Additionally, we can avoid unnecessary rotations in the distance functions by applying a filter for non-rotated shapes.

Priority Queue for Calculating Minimal Distances. As was explained in section 2.2, sphere tracing uses a variable step size in each marching step. Since this value is dynamic, we have to recalculate it in every iteration. In our baseline implementation, we do so by computing the distances to all shapes in the scene and choosing the next round's step size to be the minimum amongst them. However, this approach results in unnecessarily many invocations of the distance functions. Consider for instance the following situation with only two objects A and B in a scene. Assume that at time t we are on point p_t on the ray, and the distances to A and B are d and $4d$, respectively. In the next iteration, the algorithm continues marching along the ray with distance d to point p_{t+1}. From this point, we again calculate the distances to both objects. However, this is not necessary, as it is guaranteed that the distance to object A does not yet exceed the distance to object B.

In order to incorporate this insight, we introduce a priority queue that stores the distances to all objects at a certain iteration. Assume that at time t it is necessary to recompute all distances from the current point p_t. We store them in the priority queue in ascending order, i.e. $d_1(p_t,t) \leq d_2(p_t,t) \leq \cdots \leq d_k(p_t,t)$, where $d_i(p,t)$ denotes the distance of the i-th closest object to point p at time t. We then perform one step with step size $d_1(p_t,t)$ and land on point p_{t+1}. Next, we compute $d_1(p_{t+1},t)$, i.e. the distance of the new point to the closest object from the previous step. If

$$d_1(p_t,t) + d_1(p_{t+1},t) \leq d_2(p_t,t), \tag{2}$$

[1] More information on this profiler can be found in section 4.

then it follows directly that the closest object from time t is still the closest at time $t+1$. In that case, no further distances need to be calculated and the step size is set to $d_1(p_{t+1}, t)$. This can be repeated over multiple steps until the condition

$$\sum_{i=0}^{k} d_1(p_{t+i}, t) \leq d_2(p_t, t) \tag{3}$$

no longer holds at time time step $t+k$. At this point, it is impossible to know which object is the closest to the point p_{t+k}. Thus, all distances have to be recomputed.

Avoiding Unnecessary Rotations. Each shape has spatial characteristics described by its position and rotation. In order to calculate the SDF, the position on the ray needs to be transformed into the shape's respective object space. In our baseline implementation, the point is rotated and translated in any case, even if the shape is described by the unit rotation, resulting in unnecessary computations. By identifying objects with unit rotations in the beginning of the algorithm, we eliminate these unnecessary calculations.

With these two optimizations we drastically decrease the number of floating point operations that are performed in the code. Hence, we expect a significant decrease in this version's operational intensity compared to the baseline. In the remainder of this work, we will refer to the library version after the algorithmic optimizations as *algorithmic version*.

3.3. Mathematical Optimizations

Having eliminated algorithmic bottlenecks and thereby having reduced the number of distance function calls, we now lay the focus on applying mathematical optimizations. Our understanding of these are twofold:

i) We reduce the flop count by explicitly performing mathematical transforms and by precomputing values whenever possible.

ii) We assist the compiler in performing such transforms and in removing additional optimization blockers.

Explicit Flop-Count-Reduction. Our SPHERE library contains a lot of rendering and shading parameters that we have chosen to our taste and that are fixed across all versions. Many of these parameters are interdependent and thus share common subcomputations, or they are often used in conjunction, e.g. their product or sum is required in computations. We exploit this by precomputing such values once and storing them in new variables that are then used throughout the algorithm, rather than having to recompute them in every iteration, reducing the number of flops performed. Whenever possible, we even move these computations to compile-time using the C++ `constexpr` keyword. We move redundant (sub)computations out of loops and precompute them outside. For instance, for every pixel the origin of the ray through the pixel is at the camera lens. In the first marching step where $t = 0$, the distance to all objects has to be computed. While the baseline performed this computation for every pixel, we now only compute it once before the loop over all pixels. For a Full HD image of a scene with merely 50 objects, this already results in about 100 million less calls to the distance functions. Additionally, we perfom a great number of strength reductions throughout the entire code base.

Assisting the Compiler. While the explicit removal of redundant floating point operations shows beneficial effects, other optimization blockers still remain. Due to our OOP-based approach, functions that invoke each other are often distributed across different translation units. This is e.g. the case for the signed distance functions, residing in the translation unit of their respective shape object, which call operations on three dimensional vectors such as normalization that reside in the translation unit of the `Vector` class. Consequently, the compiler can neither inline nor apply other optimizations. In order to allow the compiler to circumvent such optimization blockers, we add the compiler flag for link-time-optimizations (LTO), effectively enabling it to optimize code across translation units.

Together with the explicit flop-count reduction, we anticipate the following impact on the overall performance: First, we expect the flop count to shrink significantly due to the explicit optimizations. Second, we presume that the `-lto` flag drastically reduces the runtime. As a consequence, we expect the performance of the algorithm to increase considerably with respect to the algorithmic version. Following our nomenclature, we hereinafter refer to this mathematically optimized version as *mathematical version*.

3.4. Vectorization

Having reduced the flop count and function calls to a bare minimum, the next logical step is to implement SIMD vectorization. For the lack of regularity in the main loops in our library (cf. section 3.6), vectorizing significant parts of the code is highly non-trivial. Although we use a priority queue to minimize the number of times the need for recalculations of the distances to all objects arises, it still happens non-negligibly often and the computational cost associated to it is large. To exploit this fact, we opt for implementing vectorized distance functions for all types of supported shapes using four-way SIMD. Consequently, we use these vectorized SDFs when having to recompute all distances, effectively allowing us to compute distances to up to four shapes of the same type at the same time.

To that end, recall that all shape objects in our code inherit from a common `Shape` base class. This means that every call to a SDF is virtual, i.e. the type of the object that calls the function is determined at runtime. Since we can

only call vectorized distance functions with objects of the same type of shape, we restructure the library such that the different shapes are grouped by type at initialization. Additionally, we add data-structures that facilitate and accelerate loading and storing to/from vector registers, and align all data on 32-byte boundaries. While this enables efficient vectorization of the distance functions, it also adds some overhead and complexity to e.g. gather the results of the different vectorized distance function calls or to differentiate between them. This is a tradeoff we expect to be worthwhile, given that the distance functions are the computationally most expensive part of the entire library.

With the above considerations, we implement vectorized versions of the distance functions. To that end, we use x86's AVX2 instructions with FMAs. Due to inaccuracies, we restrict ourselves to double precision floating point registers, a point that will be discussed more thoroughly in section 3.6.

We assume that the impact of vectorization in SPHERE is largely dependent on the scene we render. If there are very few objects in the scene, it is unlikely that vectorization is beneficial to performance. We expect that the performance benefit increases with the number of objects. However, the restriction to double precision limits the potential of vectorization in SPHERE.

3.5. Multi-Threading

Rendering our scene, we linearly iterate through an array of pixels where the colors of the pixels are only ever written once. To exploit the linearity of accesses, we implement a multi-threaded version of our main loop using OpenMP [19]. Since there are no writes to memory locations that are shared between threads, we do not require any synchronisation mechanisms or thread-safe variables. Each thread is given a set of rows of pixels to work on. However, rather than statically assigning rows to threads, we use OpenMP's dynamic scheduler for load-balancing between threads. We do this to combat differences in computational effort between rows that are inherent to the sphere tracing algorithm. To see this, consider a row of an image where the rays are shot into the void (i.e. nothing is shaded) and one row where there is an abundance of objects to be shaded. The latter will lead to more distance function and shading function calls, accumulating a significantly larger flop count.

Due to the disjoint write-access patterns, the absence of synchronisation mechanisms, and the non-existing risk for false sharing, we expect the OpenMP multi-threaded version with the dynamic scheduler to achieve an almost perfect speedup.

3.6. Other considerations

Besides the optimizations we described above, there are other optimization techniques that were considered but could not be included into our code for various reasons. We briefly describe some notheworty cases in the following.

Instruction Level Parallelism. Since we use a superscalar processor for our experiments (cf. section 4), we tried to exploit instruction level parallelism (ILP). Usually, this is done via loop unrolling and separate accumulators with the goal to fill the processor's execution pipeline as well as possible. There are two types of loops in our library for which ILP can be considered: (i) The loop over all pixels and (ii) the loop while marching on the ray. However, ILP can be applied to neither of these loops because they suffer from large irregularities for the following reasons:

(i) While the number of iterations in this loop can be determined, the loop's body can vary greatly between pixels, even for adjacent ones. While the ray through pixel (i, j) might be shot into the void, the ray through pixel $(i, j+1)$ might be reflected by objects 20 times. This all depends on the input scene, making it impossible to beneficially exploit loop unrolling.

(ii) This is a while loop that terminates when the maximum distance is exceeded or an object is hit. It is therefore impossible to foresee the number of iterations, since every iteration could be the last. Hence, unrolling is not an option here either.

Precision. AVX2 registers can either hold four double precision or eight single precision floating point values. Since our library uses double precision, a switch to single precision could theoretically bring an additional performance increase by a factor of 2 due to vectorization. In this spirit, we set up our library from the very start to facilitate a such switch of precision by using typedefs and data abstractions in the code. However, when trying to actually perform the switch in the vectorized version, our rendering program did not terminate anymore. The reason behind this is that with our default thresholds, the algorithm would run into a scenario where the step size has become extremely small. When computing the next position on the ray, an addition of the form $x' = x + s$ is performed, where both x and s can be expressed as single precision floats. However, $x + s$ cannot and is rounded down to x. The algorithm thus makes no progress and runs indefinitely. We tried to lower the distance thresholds but this resulted in outputs full of rendering artefacts. For this reason, we decided to abandon this approach.

Memory Access Optimizations. Memory objects in our library can be roughly divided into two disjoint groups: *descriptive data* and *pixel data*. The former contains objects such as the scene, its objects, and the renderer. These objects are written at initialization and are only read from then onwards. Their cumulative size is at most a few kilobytes and they thus easily fit into the cache. The latter is only

ever written once and heavily dominates the library's memory consumption, requiring about 30.5 bytes per pixel on average, regardless of the size of the scene or the version of the code. Moreover, the pixels have a perfect access pattern as they are accessed contiguously in memory. It does therefore not come as a surprise that the profiler reports barely any cache misses at all. We hence abstain from any optimizations in terms of cache access patterns.

4. EXPERIMENTAL SETUP

In this section, we specify the experimental setup we use to benchmark and evaluate SPHERE. We give an overview of the processor, the compiler and its flags, and the profiler that we used.

Processor. All of our experiments are executed on an Intel Core i7-10750H @2.6GHz processor from the Skylake microarchitecture. It supports Max Technology Turboboost 3.0 @4.8GHz which we disable for all of our experiments. Additionally, it supports AVX2 and is able to issue two FMAs per cycle, leading to a double precision SIMD peak performance of 16 flops per cycle per core. It has a three-level cache architecture where L1d, L2, L3 caches can hold 192 KiB, 1.5 MiB, and 12 MiB, respectively. The maximal memory bandwidth is specified at 45.8 GB/s. The processor has 6 physical and 12 logical cores. Since ideally a single thread running our library should be able to make (almost) full use of a core's hardware ports, we use one OpenMP thread per core for a total of six threads for our experiments.

Compiler. We compile our library using GCC 9.3.0 on Ubuntu 20.04. To profit from compiler optimizations, we use the flags `-O3`, `-march=native`, and `-ffast-math`. From the mathematical version onwards, we additionally use the `-lto` flag to enable link-time optimizations that we discussed in section 3.3. For multi-threading, we use OpenMP version 4.5 that is provided by our version of GCC. For compilation, this means setting the `-fopenmp` compiler flag and using some `#pragma omp` directives in the source code. As was already mentioned, we restrict our library to use a maximum of six OpenMP threads by setting the environment variable `OMP_NUM_THREADS` to `6`.

Profiler. In order to gather additional insights into our library and its execution, we use Intel's VTune profiler in its 2021.1.1 version, which is part of Intel's oneAPI. This allows us to gather data on the library's memory consumption, the number of cache misses, code hotspots, and many additional insightful statistics.

5. EXPERIMENTAL RESULTS

In this section, we summarize and explain our experiments and their results. First, we discuss the runtime improvements from the baseline to our final version. In the next paragraph, we analyze how the different versions of our library behave in terms of performance. Next, we perform a roofline analysis to put the observed performance into perspective by examining it relatively to our benchmarking machine's peak performance. These three experiments use the same input scene that is depicted in figure 1. There are 96 shapes in this scene in total, 16 of each shape type. However, only 24 of them are visible. We conclude this section with an investigation on the impact of the number of objects in the scene on the performance of the vectorized version.

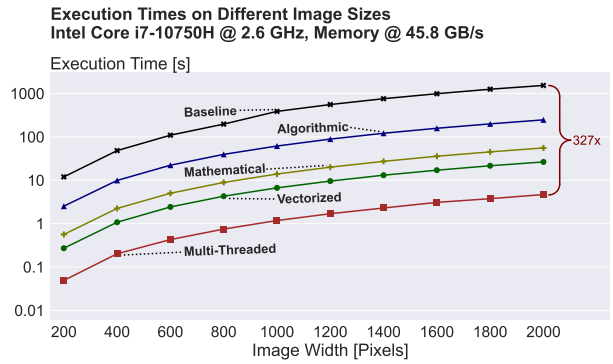
5.1. Speedup Analysis

In this first experiment, we examine the speedups that our optimized code versions achieve on the default benchmarking scene for a variety of different image sizes, ranging from very small (200×100 pixels) to approximately Full HD resolution (2000×1000 pixels). The results are depicted in figure 4a. Note that the y-axis is scaled logarithmically. The first thing to notice is the respectable speedup of 327 from the baseline to the final, multi-threaded version. This speedup stays more or less constant across all image sizes. The same holds for the speedup between any two versions of the library.

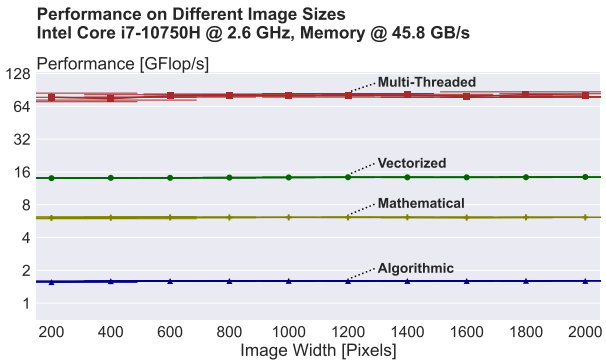
The algorithmic version considerably reduces the runtime compared to the baseline implementation. This is overwhelmingly due to the priority queue (cf. section 3.2). It is in line with our expectations since through these optimizations we are able to reduce the number of distance function calls by orders of magnitude.

From the algorithmic to the mathematical version, we extract a speedup of 4.4. Most likely, a large fraction of this speedup can be justified with the removal of the optimization blockers. Function inlining plays a significant role here: The majority of the flops are carried out as a part of operations on the three-dimensional `Vector` object, whose operators are all overloaded and reside in their own translation unit. These can be inlined in the mathematical version, while this is not the case in the algorithmic version. Furthermore, this enables our explicit mathematical transforms to come to full effect.

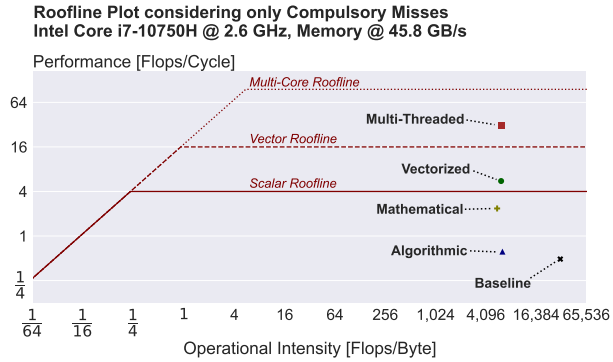
Introducing double precision AVX2 instructions could in theory result in a speedup of up to four. However, on our benchmarking scene we only achieve a speedup of 2.1 with respect to the mathematical version. We explain this discrepancy as follows: Firstly, we only vectorize the distance functions, and this vectorized version is only called when all distances have to be recomputed. Secondly, the introduction of new data structures to enable grouping the shapes by type necessary for the vectorization of the distance functions adds additional overhead in the form of `if-else`-branches to distinguish between the types of the shapes, which are harder to predict for the branch predictor. In con-



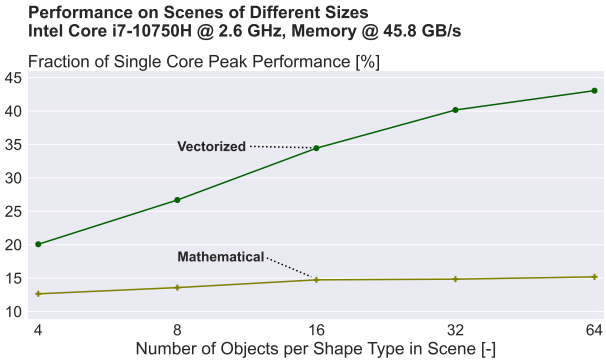
(a) Execution time of all code versions across input sizes. We extract a total speedup of 327 over the baseline.



(b) Performance of different versions across input sizes. The baseline is not included due to its significantly different flop-count.



(c) Roofline plot for all code versions. We provide rooflines for scalar, vector, and multi-core execution.



(d) Comparison of the vectorized and mathematical version on different scene sizes.

Fig. 4: Experimental results for SPHERE when evaluated on some fixed benchmarking scenes.

trast, the mathematical version uses virtual function calls instead of branches.

The speedup delivered by the multi-threaded version accords with what we have anticipated. Using six threads, we achieve a nearly perfect speedup of 5.7 (out of a maximum of 6). The small difference comes from parallelization overheads from the scheduler. Note that we want to stress here that, as described in section 3.5, the near-perfect speedup is only possible with dynamic scheduling. Static scheduling merely achieves a speedup close to 4.5.

5.2. Performance Analysis

In the next experiment, we are interested in the performance that our different versions achieve. Recall that the algorithmic optimization drastically reduced the flop count over the baseline (we will quantify the size of this reduction in section 5.3). For this reason, the baseline is omitted in this experimental setting. The performance plot in figure 4b shows the performance that the different versions achieve on varying image sizes.

Note that for all versions the performance is independent of the image size. This constancy can be explained by the fact that the code is heavily compute bound. As such, the input size neither affects the number of cache misses nor the optimality of the memory access pattern. Together with a linear increase (in the image size) of the number of flops, which is to be expected as the number of shapes remain the same, this results in the constant performance. Comparing the mathematical and vectorized version, we observe that the performance gain and the decrease in runtime (figure 4a) do not correspond. In particular, the runtime decreases by a factor of 2.1 while the performance increases by a factor of 2.3. To understand this, we have to look at the infrastructure changes that we undergo from the mathematical to the vectorized version. Not only does the additional logic for vectorization introduce new logical complexity in the code surrounding the vectorized parts as outlined in section 3.4, but rewriting the distance functions with AVX2 instructions also leads to an increase in floating point operations. This is due to the fact that through the elimination of `if-else`-statements, each branch has to be executed in all vectorized functions (possibly with masks), introducing additional floating point operations and prohibiting early return statements. Furthermore, the absence of branching required for

performant vectorization forces us to calculate every rotation even if the object we are calculating the distance to is rotated by the unit rotation.

Summing up the improvements over all optimized versions, we can report a significant increase in performance of a factor of almost 51 from 1.6 GFlop/s in the algorithmic version to 81 GFlop/s in the multi-threaded version.

5.3. Roofline Analysis

To understand the correlation between performance, flops, and memory consumption, we perform a roofline analysis which is shown in figure 4c. For reasons described in section 3.6 we only consider compulsory read and write misses. We plot the rooflines for single-core scalar and vectorized execution as well as for multi-core vectorized execution. The memory footprint of our library is measured with Intel's VTune profiler. Evidently, all of our versions operate in the compute-bound regime. This does not come as a surprise since the task that our library carries out requires comparably few memory reads/writes. In the roofline plot, we observe a significant decrease by a factor of about 5 in operational intensity from the baseline to the algorithmic version. This underlines our argument not to include the baseline in the performance plot. For the other versions, the operational intensity remains about the same whereas the performance increases with each optimization step.

For the benchmarking scene, the mathematical version reaches almost 59% of non-vectorized single core peak performance, the vectorized version 34.6% of vectorized single core peak performance, and the multi-core version 32.5% of vectorized multi-core peak performance. The drop in percentage of peak performance from the mathematical to the vectorized version directly follows from the non-optimalities of the vectorization described in the previous paragraphs.

5.4. The Effect of the Scene Size

In the aforementioned experiments we have seen that the vectorized version cannot achieve its full potential on the default benchmarking scene. We therefore analyze the impact of the number of objects in the scene on the performance of the vectorized version. Figure 4d shows that the number of objects in the input scene considerably influences its achieved fraction of peak performance. As we increase the number of objects per shape type in the scene, the vectorized version gains in performance, increasing from 20% to almost 45% of single-core machine peak performance. To put this into perspective, the mathematical version only enhances its fraction of (SIMD) single-core peak performance from 13% to about 15%. We justify this discrepancy in the performance gains as follows: With the increase in the scene size, the number of calls to the vectorized distance functions rises disproportionally fast compared to the number of calls to the functions controlling the colouring of pixels and to non-vectorized distance functions. Consequently, larger fractions of the executed code are vectorized, leading to a steady increase in performance. As the scene size increases, we expect the vectorized performance to keep rising until it hits a theoretical upper bound. Given that the vectorized single-core peak performance of the mathematical version flattens at around 15% (corresponding to 60% of non-vectorized single-core peak performance), we conclude that the performance of the vectorized version is bounded from above by 60% of vectorized single-core peak performance.

6. CONCLUSION

In this work, we implement a heavily CPU-optimized sphere tracing algorithm. We derive a baseline version and successively apply thematic optimizations such as algorithmic optimizations, mathematical optimizations, and vectorization. These improvements are inspired by bottlenecks found by Intel's VTune profiler, a tool we use extensively across all versions. While many of the achieved experimental results conicide with our expectations, the vectorization does not always achieve its full potential. By analyzing different input scenes, we show that the performance of vectorization is heavily dependent on the number of shapes in the input scenes. Increasing this number, we reach 45% of vectorized peak performance on our benchmarking machine, an achievement we deem respectable, considering that the library is only partially vectorized. In total, we achieve a speedup of 327, and increase the performance from 1.6 GFlops/s to 81 GFlops/s on our default benchmarking scene. This allows us to generate a high quality rendering of a complex scene in a few seconds. We conclude that, contrary to common belief, it is feasible to produce high quality renderings of sophisticated scenes using only a CPU in a reasonable amount of time.

7. CONTRIBUTIONS OF TEAM MEMBERS

We summarize the contribution of the individual team members as follows:

Andre. Mathematical optimizations in main loop and render, vectorizing parts of distance functions. Responsible for collecting data for plots. Multithreading.

Christopher. Algorithmic optimizations, i.e. introducing priority queue data structure, vectorized parts of distance functions. Implemented data collection and plotting functions.

Jens. Compiler help implementation, making LTO work and introducing data structures for vectorization. Responsible for design of plots. Tried header only library and removal of virtual function calls. Multi-threading.

David. Mathematical optimizations in all distance functions. Vectorized parts of the distance functions. Algorithmic optimization, i.e. removing unnecessary rotations.

8. REFERENCES

[1] Jörg Schmittler, Daniel Pohl, Tim Dahmen, Christian Vogelgesang, and Philipp Slusallek, "Realtime ray tracing for current and future games," in *Informatik 2004 – Informatik verbindet – Band 1, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Peter Dadam and Manfred Reichert, Eds., Bonn, 2004, pp. 149–153, Gesellschaft für Informatik e.V.

[2] Per H. Christensen and Wojciech Jarosz, "The path to path-traced movies," *Found. Trends. Comput. Graph. Vis.*, vol. 10, no. 2, pp. 103–175, Oct. 2016.

[3] Bogdan Deaky and Luminita PARV, "Virtual reality for real estate – a case study," *IOP Conference Series: Materials Science and Engineering*, vol. 399, pp. 012013, 09 2018.

[4] John E. Stone, *Interactive Ray Tracing Techniques for High-Fidelity Scientific Visualization*, pp. 493–515, Apress, Berkeley, CA, 2019.

[5] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer, "The randomized z-buffer algorithm: Interactive rendering of highly complex scenes," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2001, SIGGRAPH '01, p. 361–370, Association for Computing Machinery.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter, "Distributed ray tracing," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1984, SIGGRAPH '84, p. 137–145, Association for Computing Machinery.

[7] J. C. Hart, D. J. Sandin, and L. H. Kauffman, "Ray tracing deterministic 3-d fractals," *SIGGRAPH Comput. Graph.*, vol. 23, no. 3, pp. 289–296, July 1989.

[8] John Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, 06 1995.

[9] NVidia Corporation, "RTX. It's On. Ultimate Ray Tracing And AI," `https://www.nvidia.com/en-us/geforce/rtx/`, Online; accessed 06/25/2021.

[10] André Gaillard, Christopher Raffl, Jens Eirik Saethre, and David Spielmann, "SPHERE Library," `https://gitlab.inf.ethz.ch/COURSE-ASL/asl21/team23`, June 2021, Online; accessed 06/25/2021.

[11] Albrecht Dürer, *Underweysung der Messung, mit dem Zirckel und richtscheyt, in Linien, Ebnen und gantzen Corporen*, Formschneider, Nürnberg, 1538.

[12] Turner Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, June 1980.

[13] James T. Kajiya and Brian P Von Herzen, "Ray tracing volume densities," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1984, SIGGRAPH '84, p. 165–174, Association for Computing Machinery.

[14] Dario Seyb, Alec Jacobson, Derek Nowrouzezahrai, and Wojciech Jarosz, "Non-linear sphere tracing for rendering deformed signed distance fields," *ACM Trans. Graph.*, vol. 38, no. 6, Nov. 2019.

[15] Eric Haines and Peter Shirley, *Ray Tracing Terminology*, pp. 7–14, Apress, Berkeley, CA, 2019.

[16] scratchapixel, "Rendering implicit surfaces and distance fields: Sphere tracing," `https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields/introduction`, Online; accessed 06/25/2021.

[17] Inigo Quilez, "Distance functions," `https://iquilezles.org/www/articles/distfunctions/distfunctions.htm`, Online; accessed 06/25/2021.

[18] Bui Tuong Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, pp. 311–317, June 1975.

[19] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," November 2015.