A HIGHLY OPTIMIZED CPU RENDERING ENGINE USING RAYMARCHER BASED ON SPHERE TRACING

Nihat Isik, Qais El Okaili, Alexandre Cavaleri, David Graf

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We implement a rendering engine using the Ray Marcher method based on Sphere Tracing, a well-known technique for rendering implicit functions. This method is used in computer graphics for a variety of effects [1]. Although fast implementations are usually programmed on the GPU, we focus optimization on CPU by employing a systematic approach summarized in four phases. Eliminating memory-bound issues and unnecessary computations initially and after leveraging vectorization we obtain a speed-up of $32.1\times$. Finally, we suggest the use of multithreading to extend our implementation for real-time rendering.

1. INTRODUCTION

Motivation. Ray Marching is a rendering technique, similar to Ray Tracing, used to compute 2D images from a 3D scene. Although a traditional Ray Tracing is able to render complex scenes in a fraction of a second, it is limited regarding volume rendering and complex mathematical shapes by its inherent use of ray intersections with polygons. A Ray Marcher opens up the possibility to render infinitely detailed mathematical objects like fractals, spheres, or a torus without the need for an infinite amount of polygons. Furthermore, mathematical operations can be added to any object allowing infinite duplication, merging, or even real surface displacement. Unfortunately, Ray Marching is considerably slower than Ray Tracing for simple polygon meshes and thus, is only seldom used. Despite this challenge, Ray Marching remains a powerful algorithm used to render clouds, fog-light interaction, or ambient occlusion in modern AAA video games, where each frame needs to be rendered in a few milliseconds. While ray marching each pixel can be easily parallelized, marching along a ray for a single pixel is inherently sequential. There are only a few data re-use in the process and the execution time for different pixels can vary drastically depending on their location.

Contribution. In the context of the ETH Advanced Systems Lab course, we focused on optimizing the rendering pipeline on the CPU without the use of parallelism. Our first contribution is a rendering engine based on Ray Marching using Sphere Tracing which can display visually complex scenes containing fractals, spheres, cones, or infinite cubes with reflections and soft shadows. Our second contribution is the identification and optimization of major bottlenecks. And this in a guided approach across multiple phases. Finally, the optimized rendering engine is able to render the exact same scene, with all features, up to $32.1\times$ faster than the baseline.

Related work. Hart et al. introduced a technique to render implicit surfaces using geometric distances [2]. This approach to Ray Marching proves to make the algorithm converge faster than traditional methods. While we based our baseline on the Sphere Tracing idea, we focused our contribution on increasing performance in place of algorithmic enhancements. Inigo Quilez wrote numerous articles on his blog [3], which have become the reference for the more complicated mathematical and implementation details of a Ray Marcher. Even though his code snippets are designed to be run on GPU shaders it helped us adapting the mathematical concepts to the CPU microarchitecture. Finally, the only fast CPU implementation we found is a C++ Ray Marcher based on Sphere Tracing [4]. In this implementation, the author utilized SIMD vectorization and multithreading to achieve a fast rendering of a scene containing only cubes. Our final optimized Ray Marcher was not only able to beat this implementation by a factor of $14\times$ for the same recreated scene [5], but also provides many more shapes and optimized features.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

Rendering a scene consists of shooting a ray for each pixel of the image and shading the closest intersection point between the ray and the scene. In the traditional way of rendering, each shape is composed of multiple primitive shapes such as triangles, that simplify the computation of a ray intersection. However, for more complex shapes such as fractals, this method does not work. This is exactly where Ray Marching shines. The main idea of a Ray Marcher is to

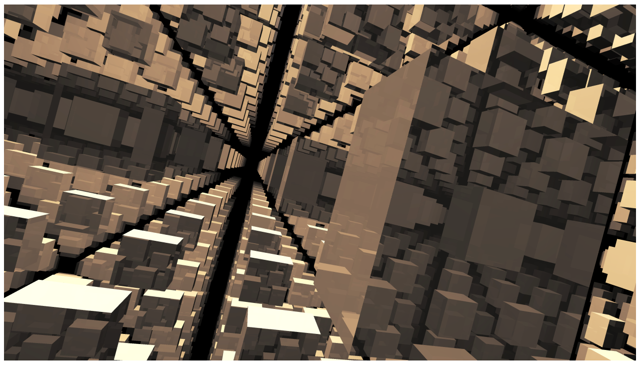


Fig. 1: An example rendering of our program of the infinite cube repetition with reflective material.

render *implicit functions*. These are functions of the form

$$F(x, y, z) = 0, \quad F : \mathbb{R}^3 \longmapsto \mathbb{R}$$

that are called Signed Distance Functions (SDF). They provide useful inside-outside information via their sign. Given a point $p = (x, y, z)$ and an SDF shape $F(x, y, z)$, then

$$p \text{ is inside } F \text{ iff } F(p) < 0$$
$$p \text{ is outside } F \text{ iff } F(p) > 0$$

In order to get the intersection point between a ray and the shape, we march along this ray until the SDF value becomes smaller than a threshold (or negative). In our case, we define the marching algorithm via *Sphere Tracing*. This algorithm guarantees to never miss an intersection by marching along a ray using the distance to the closest object in the scene. Below, we show the pseudo-code for the rendering engine. Figure 1 shows an example output of our rendering engine.

Algorithm 1: Rendering Pipeline

```
Data: Scene.
Result: Rendered image based on input scene.
for each x,y pixel do
    p, dir = (orig, getDir(x,y));
    sphereTracing(p, dir, Scene);
    if not intersected then
        pixel[x,y] = ambientColor;
        continue;
    end
    if reflective then
        recursiveTrace(Scene);
    end
    pixel[x,y] = computeMaterialShading();
end
```

Cost Analysis. We counted the floating-point operations (FLOP) manually, based on the assumption of one FLOP for each math.h function. We tracked the count for each function in our code in an excel spreadsheet. Then, we instrumented our code to count the number of calls of each function. Finally, we multiply each function call count by its respective FLOP count to get the final cost of our Ray Marcher. We validated our function call counts using the Intel Software Development Emulator (SDE). In Table 1 we give the FLOP count for the two scenes we benchmarked and profiled throughout the project.

Algorithm 2: Sphere Tracing

```
Data: Origin point p, Ray Direction dir, Scene.
Result: Intersection point of ray with scene.
for k raymarching steps do
    min_dist = INF;
    for t shapes in scene do
        min_dist = min(min_dist, distanceToShape(p, t));
    end
    p = march from p by min_dist in direction dir;
    if $min\_dist < eps$ then
        break;
    end
end
```

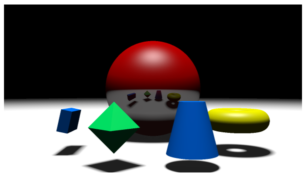
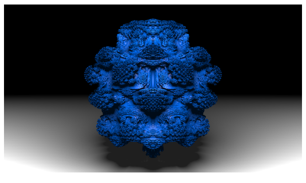
Scene0	**Mandelbulb**
20.68 $GFLOP$	54.35 $GFLOP$
	

Table 1: Floating-point count for scene0 and mandelbulb for $n = 720$ (resolution 1280×720).

The Mandelbulb scene is computationally more intensive with more than twice the FLOP count. This imbalance causes a higher percentage of peak performance for the Mandelbulb scene because it is spending more time on computations than scene0 which has the overhead of switching between SDF shapes. The runtime of the Ray Marcher depends a lot on the content of the scene. For example, in our baseline, the Mandelbulb takes around 12% less time than scene0, and rendering shapes like a single cube is much faster. The asymptotic complexity of our Ray Marcher is $\mathcal{O}(KTN^2)$: Where the width and height values are simplified to N. T is the number of shapes and K the maximum number of marching steps. However, the mentioned complexity is rarely reached because most of the rays intersect an object or reach the bound of the scene long before K steps is reached. It can mostly be reached in pixels containing edges where the marching step size is heavily reduced due to the close proximity.

3. OPTIMIZATION PHASES

We methodologically addressed bottlenecks in the Ray Marcher and optimized them in phases. In the first phase, memory-boundness was prominent, and we reduced it. From the second phase on we mainly faced computation-bound problems. In the third phase, we started SIMD vectorizing the rendering engine and got a significant speed-up. In the fourth and last phase, we made further improvements to SIMD vectorization and achieved higher performance.

We used Intel VTune to profile our program. The microarchitecture usage is an estimate of how effectively our code is using the microarchitecture, in other words, how much further we can optimize. This percentage measures how front-end bound (instructions overhead), memory-bound and computation/core bound the program, or any given function is. Throughout this paper, when we refer to a specific bound percentage, it represents the portion of pipeline slots (hardware resources) remaining empty due to the bound. The retiring percentage, on the other hand, represents the portion of pipeline slots utilized by useful work.

3.1. Phase 1

The hotspot analysis of our baseline listed in Table 2 shows the rotation transformation applied to each object in the scene is the biggest bottleneck. In more detail, the *__sincos* function turned out to be the first major barrier. The next major bottleneck is the vector library operations such as *vec_norm* and *vec_sub*. Finally, we identified the square-root operation as one of the primary bottlenecks in many SDF shapes and in the *vec_norm* function. In what follows, we give a detailed analysis and our optimization strategy for the mentioned bottlenecks.

Function	Time	Usage
apply_trans.	$8.96s$	69%
__sincos	$3.07s$	15%
vec_norm	$2.76s$	27%
vec_sub	$2.73s$	29%
sdf_cone	$2.55s$	34%
sdf	$2.05s$	40%
new_vector	$1.47s$	65%
sdf_torus	$1.42s$	13%
sdf_octah.	$1.13s$	25%
rotate_point	$1.04s$	50%

(a) Phase 1 (On Intel i7-7500U, 2.70 GHz)

Function	Time	Usage
sdf	$2.52s$	69.7%
sdf_cone	$2.02s$	54.1%
vec_norm	$1.77s$	42.3%
sdf_octah.	$0.90s$	18.1%
sdf_torus	$0.78s$	57.1%
vec_sub	$0.63s$	18.9%
ray_march	$0.50s$	46.3%
sdf_box	$0.46s$	17.4%
vec_dot	$0.42s$	100.0%
sdf_plane	$0.39s$	39.0%

(b) Phase 2 (On Intel i7-4930k CPU, 3.40GHz)

Table 2: VTune Hotspot analysis of our baseline in Phase 1 shown in (a) and of master_opt1 in Phase 2 shown in (b). The table shows the CPU **Time** and the Microarchitecture **Usage** of the top hotspot functions.

Transform Operations. An extensive microarchitecture analysis showed that a significant amount of pipeline slots were remaining empty due to long latency memory operations and, to a smaller extent, due to computations. In more detail, the __sincos assembly analysis showed that a large percentage of time was spent on *movl* instructions (memory-bound). We approached the optimization of this part by relying on certain assumptions: First, the object's rotation does not change while rendering a frame (Static Scene). Second, most objects have a zero rotation transformation. Thus, we only evaluate rotations when their values are non-zero.

Due to the first assumption, we precomputed rotation values, using *sin* and *cos*, effectively reducing function calls to only one. This led to a speed-up of $2\times$, as a result of lower memory and computation-bound. Due to the second assumption, we only execute rotation code on a non-zero angle, which reduced the number of calls and results in an extra runtime speed-up of $1.2\times$. Finally, by inlining the function and removing unnecessary data copies we achieved an extra runtime speed-up of $1.1\times$.

Vector Library. The vector library consists of common operations like the dot-product and vector addition. In the baseline, we use a *struct*, called *Vec3*, with three *double* elements to represent a vector. Most of the vector operations have a *Vec3* return type, for which we call the *new_vector* function that returns a *Vec3* copy allocated on the stack. This turns out to be a significant overhead because of the high usage of these operations. The majority of the vector operations are bounded by the L1 Cache and in many cases, this is due to loads blocked on older stores in the CPU pipeline.

Also, port utilization is low, and more detailed analysis showed us that in some cases this was due to data dependencies between nearby instructions. The vector operations are modified to return pointers instead of explicit types. This avoids the usage of the new_vector function and as a result, reduces stack-allocation. In addition, every input parameter is passed by reference instead of a copy. These vector optimizations granted a runtime speed-up of $1.25\times$.

Square root. In Table 2 the vec_norm function appears to be a bottleneck, this is not only due to the aforementioned vector problems, but prominently because of the square root computation. We use the *sqrt* function from the math.h library that returns an exact square root computation. It is not only used for computing the vec_norm but also appears in most of the SDF shapes and is computationally expensive to execute. Our main approach was to use the *fast inverse square root*[6] algorithm, the *newton step method* [7] and the SSE $RESQRT\_ps$ [8] intrinsic to approximate *sqrt* . However, we were not able to get satisfying results with these approximation methods as high approximation errors affected heavily the output image.

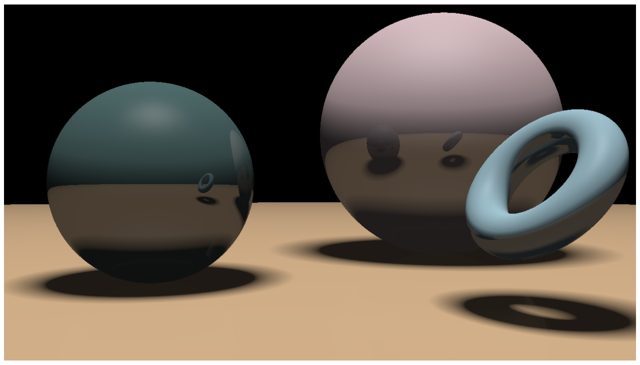


Fig. 2: An example output rendering of our program with two spheres and a torus with reflective surfaces.

3.2. Phase 2

Performing a new hotspot analysis, shown in Table 2b, after the optimizations from phase 1, we noticed that the main bottlenecks are now caused by the different SDF shape functions. Once more, our vector library and the *sqrt* operation show up in the top hotspots. In Table 2 an analysis of this phase is depicted. In the following, we present our optimization for the bottlenecks just mentioned.

SDF Functions. A microarchitecture analysis of *sdf* code showed us that the *sdf* function was badly utilizing the pipeline due to heavy computations like divisions and cache misses causing stalls. The profiling results showed that the SDF function is both memory and computation-bound. This is caused by overloaded execution units and chained long latency arithmetic operations with dependencies between them. Effectively, the divider unit is only active for a significant portion of the execution time. More importantly, 46.5% of cycles have zero port utilization due to the long latencies incurred by divisions and cache misses. Looking at each *sdf* function more in detail we noticed that many redundant data copies and computations are performed. In general, the SDF shapes are highly computation-bound. The basic shapes such as plane, sphere, and torus mostly bound by the *vec_norm* function because of the square root computation.

Precomputing redundant calculations made at each SDF call lowers the computation load of the SDFs and frees up the divider unit that was causing stalls in the pipeline. Furthermore, we inlined most of the functions and lowered the number of operations. For example, if one entry of a vector is zero, we could remove the operation for that entry in case of vector multiplication, addition, and so on. Additionally, we tried to unroll the *SDF loop* by a factor of two with two accumulators in order to break the sequential dependency. However, this didn't increase performance. Further optimization was to lower bad speculation and unnecessary condition checks in the SDF shapes by ordering the conditions by their call count. Additionally, we managed to transform some branches that handled the sign of a value into the expression $1 - 2 * condition$, where $condition$ is either zero or one.

SDF Mandelbulb. The Mandelbulb fractal SDF is mainly compute-bound but also has some memory-bound issues, mainly L1 bound (21.3%). Also, many computations were simplified analytically and therefore reduced the FLOP count.

Unrolling the entire for-loop in the Mandelbulb code allowed more data reuse. In addition, many computations could be simplified. As an example, $\sqrt{a^7}$ using six multiplications could be reduced to $a^3\sqrt{a}$ using three multiplications. Furthermore, reused factors or multiplicands were precomputed and code motion was done to simplify some expressions.

As a result, the performance increased by 30%, and the memory-bound was lowered from 21% to 13%, for a total runtime speed-up of $1.38\times$.

Vector Library. The second hotspot analysis showed that the vector operations were still a major bottleneck. Upon further investigation, we found that the *Vec3 structs* as a form of data abstraction for all our vectors was causing overhead in the front-end.

Bound	**Fraction**
Front-End	31%
Memory	10%
Retiring	41%
Core	18%
Runtime	**13s**

(a) Structs

Bound	**Fraction**
Front-End	23%
Memory	5%
Retiring	52%
Core	16%
Runtime	**10s**

(b) Arrays

Table 3: Vector optimization: in phase 1 we use *structs* (a) and in phase 2 we use *arrays* (b). The listed results are for rendering an image resolution of 1280×720.

We shifted from *structs* to size three *arrays* to represent our vectors. Table 3 shows on the left the Bound metrics on phase 1 before the array optimization and on the right the outcome of using arrays. All the metrics have improved much and the runtime speed-up is $1.3\times$.

Square root. The optimizations performed in phase 1 highlighted even more the *square root* bottleneck during profiling. Approximating the *square root* again, achieved valid outputs on the condition of increasing the precision by adding a Newton iteration step for the *fast inverse square root*. However, this decreased performance of the *square root* and thus was not useful.

Instead of optimizing the square root, we started focusing on reducing the number of times it is called. For example, ray boundary detection, which just compares the norm to a constant could be easily modified to compare with the squared constant and thus avoid a costly square root operation at each ray marching iteration.

3.3. Phase 3

Up to this point, we performed multiple optimizations to each function in our pipeline considering the most common techniques like code motion, loop unrolling, precomputation, simplifying expressions, and so on. However, we were approaching a barrier with these types of optimizations. Table 4a shows that the main bottlenecks are still inefficiently using the hardware. At this stage, our program was not bound by memory anymore, but mainly computation-bound. A microarchitecture analysis on our Ray Marcher showed that the part of pipeline slots remaining empty was 9% due to load or store instructions and 24.8% due to computations. Thus, we focused on what could make computations more performant: SIMD vectorization. Below, we suggest an approach to vectorizing our Ray Marcher.

Function	**Time**	**Usage**
sdf_cone	$2.10s$	35.0%
vec_norm	$2.05s$	62.2%
sdf	$1.23s$	31.8%
sdf_octah.	$1.02s$	16.1%
rotate_point	$0.43s$	100.0%
vec_sub	$0.42s$	26.4%
sdf_torus	$0.26s$	34.9%
sdf_sphere	$0.24s$	100.0%
clamp	$0.22s$	62.9%
ray_march	$0.22s$	50.0%

(a) Phase 3

Function	**Time**	**Usage**
sdf	$0.80s$	41.5%
sdf_cone	$0.60s$	30.5%
vec_norm	$0.59s$	44.7%
ray_march	$0.26s$	30.3%
sdf_octah.	$0.18s$	69.9%
vec_sub	$0.15s$	72.8%
sdf_box	$0.13s$	54.2%
vec_norm_sq	$0.13s$	65.1%
sdf_torus	$0.07s$	65.1%
rotate_point	$0.06s$	46.6%

(b) Phase 4

Table 4: VTune Hotspot analysis of master_opt2 in Phase 3 shown in (a) and of master_opt3 in Phase 4 shown in (b), on Intel i7-4930k, 3.40GHz. The table shows the CPU **Time** and the Microarchitecture **Usage** of the top hotspot functions.

Vectorizing our pipeline means shooting multiple rays simultaneously. This means that we have to keep track of the state of all the rays when tracing them, even when calling functions like *ray_march* and *sdf*. On top of this, we need to make sure that the recursive trace function works properly and the cases where a ray returns early be accounted for without interrupting other rays. To make all this work properly we first had to come up with a good representation of our vectors using SIMD variables.

Horizontal vector representation. Our initial approach was to represent each vector by one SIMD *m256d* variable, which would contain the three vector elements x_0, y_0 and z_0 and a redundant unused entry. This choice was motivated by the similarity to our baseline vector representations. An advantage is the straight-forward and ease of integration. Disadvantages are the one unused entry in the SIMD variable and flattening operations, e.g., for the dot-product and vector norm function (need of *hadd* or similar instructions).

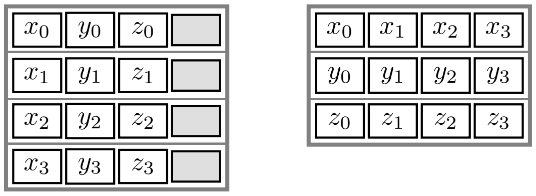


Fig. 3: The left figure shows a horizontal layout where each vector is represented as one SIMD *m256d* variable with an unused entry. The right figure depicts a vertical vector layout using three SIMD *m256d* variables that can pack four vectors.

Vertical vector representation. In the second approach, we changed the representation to be vertical. Our new SIMD vector data structure is encapsulated in a struct named *SIMD_VEC* which consists of three SIMD *m256d* variables that we name **x**, **y** and **z**. The *m256d* variable **x** includes the first four elements of the four vectors, i.e., $\mathbf{x} = \{x_0, x_1, x_2, x_3\}$, respectively the same for **y** and **z**. Thus, we can encapsulate four vectors in one *SIMD_VEC* data structure. The disadvantage here is the change of structure of how our operations are performed. However, the advantages include that we do not waste space. We can shoot and process four vectors while performing only three SIMD intrinsic operations instead of four. Furthermore, no flattening and no additional overhead. Thus, we chose to adopt the vertical representation to shoot and process four rays simultaneously.

Vectorizing Trace, Ray March and SDF Functions. First, we unrolled the inner loop in the *render* function by a factor of four considering also the residual loop for a *width* non-divisible by 4. Then we changed the *shoot_ray* function so to process four pixels and return a *SIMD_VEC* containing the ray directions. The origin point is just another variable initialized to be zero.

Vectorizing the *trace* was not very complicated after having the input as a *SIMD_VEC*. Most of the computation had to be repeated for the four vectors. However, the *trace* function also contains more complex functions and recursion which was a bit more cumbersome to deal with. This is due to the fact that even if only a single ray hits a reflective object the recursive function call has to happen and has to be filtered only for that single ray.

Thus, we start by creating a mask to mark the reflective rays. Then, we call *trace* recursively passing as input only the reflected rays and zero out the others. Finally, when the color value is returned we mask out the color values for non-reflective rays and sum this value to the final color.

Vectorizing the *ray_march* function required us to use a lot of logic operations and masking techniques learned in class. Since we know for a fact that each ray has to be terminated in order to exit this loop, we have to keep track of the state of each ray independently from the others and be able to filter state changes. This function heavily depends on the *sdf* function which returns the distance to the closest object in our scene. The ray state (i.e. if ray finished marching or not) is passed from the *ray_march* function to the *sdf* function which uses it to do masking operations.

Thus the *sdf* function changed in the following way:

1. apply the transform for the 4 input points
2. compute distance to the closest objects for the 4 points
3. update *min_dist* vector iff ray is not in finished state
4. do the same for the nearest object index (identifier used to get material of closest object)
5. **repeat from 1** for the next shapes **but** step 3 becomes: update the *min_dist* vector iff we found *closer* distances and the ray is not in finished state

Next comes the vectorization of the *ray_march* function that we summarize in the following list:

1. initialize and precompute all needed variables
2. call *sdf* function to get the distances to the closest shapes
3. build intersection mask (*distance* < *Threshold*) and use previous overshoot mask to build the mask of finished rays (= OR($intersect\_mask$, $overshoot\_mask$))
4. filter the current *min_distances* to not contain finished rays and march along unfinished rays
5. if the intersection mask changed from the previous mask update $current\_intersect\_mask$
6. compute new overshoot mask, if it changed from the previous mask update $current\_overshoot\_mask$
7. check exit condition: if all rays are finished then exit

Finally, we applied additional improvements to exit early when checking for the changes in steps 5 and 6. This allows us to spare some *MOVEMASK* and other SIMD logic operations.

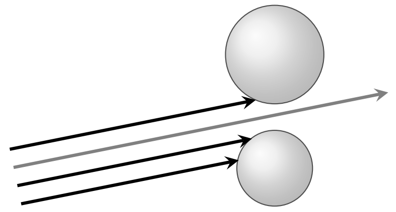


Fig. 4: Vectorizing ray marcher function. Depicted are four rays marching simultaneously and while some rays have already intersected an object, others are still marching.

Vectorizing simple SDF Shapes. Vectorizing plane, sphere, torus, and the cone was a straight-forward conversion from scalar code to SIMD code. For the box SDF, we had many ternary instructions to compute the absolute value. Using a simple trick with *vandnpd* instruction we can force the sign bit to zero and thus get the absolute value with fewer operations.

Vectorizing Octahedron SDF. Vectorizing the octahedron SDF required a more careful analysis of the code since it is composed of multiple if-else branches. We are operating on four points and this implies some changes in the ordering of the branches and the exit condition. First, we can enter the exit branch only if all four points land into that branch. To do this, a mask is created for each branch using *CMP* instruction. Then, the *MOVEMASK* instruction is used to check if all points can safely return from the function. If this is not the case, then apply the computations for each branch using the respective masks combined with *ANDNOT* and *BLENDV* instructions.

Vectorizing Mandelbulb SDF. The sequential Mandelbulb SDF has a for-loop of length four that is unrolled completely in Phase 2. In each unrolled loop $i \in \{0, 1, 2, 3\}$ a computation c_i is done resulting in a variable m_i. At the end of each loop, a check is made, if m_i is greater than 256 then we return some final computed value. Vectorizing each computations c_i was straight-forward. However, handling the variable returns for each ray simultaneously needed careful handling. As depicted in Figure 5 after each computation c_i a return evaluation on the rays is done via comparisons and maskings. If a ray returns, it is marked in a mask and due to this, its later computations in the next loops will not be added up anymore. This is done until all rays have returned.

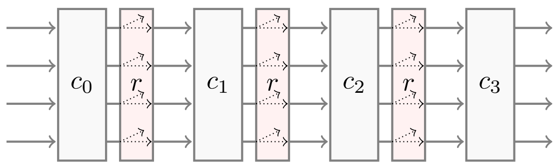


Fig. 5: Vectorization of Mandelbulb SDF. Each ray depicted as an arrow can return in r after each computation c_i or continue to the next computation c_{i+1} for $i \in \{0, 1, 2, 3\}$.

Approximation of Log and Exp. Intel intrinsics doesn't provide vectorized instructions for operations such as *log*, *exp* and *pow* for GCC. An alternative solution is to use approximation. We found a good Min-Max polynomial fit approximation from the Mesa 3D source code[9]. However, the provided approximations use single-precision instructions, so we converted the code to fit our purposes.

All the above-mentioned changes gave birth to a fully SIMD vectorized rendering pipeline working on all our machines and giving us a significant performance increase.

3.4. Phase 4

The SIMD vectorization result of our Ray Marcher in Phase 3 lowered the computation-bound considerably. The percentage of pipeline slots remaining empty due to memory issues rose to 32% and due to non-memory issues was reduced down to 20% on an i7-4600U CPU. Thus, we focused on an approach that could lower the memory requirements and the computation load. We switched the Ray Marcher program from using 64-bit doubles to 32-bit floats. This change allowed taking advantage of single-precision SIMD instructions and shoot eight rays instead of four. Thus, executing twice the number of computations simultaneously, while having almost the same number of instructions. Single precision floating points caused some artifacts which required careful parameter fine-tuning to have an exact match with the previously rendered images.

3.5. Phase 5

In this added phase, we further improved our Ray Marcher to exploit the parallelism of the problem. With the desire to compare it to a real-time CPU Ray Marcher called CPU-RTRM[4] which uses multithreading. Splitting the image so that subgroups of pixels are traced on different threads led to a speed-up of 2.67× from the previous phase. Analyzing the time spent on rendering alone allowed us to compare with CPU-RTRM and observe a 14x speed-up for the same scene. We did not include this phase in the results since, in the context of the ASL course, it is out of the scope of the project.

4. EXPERIMENTAL RESULTS

In this chapter, we discuss the results of the baseline and each optimization phase. We built a benchmarking infrastructure that uses the RDTSC instruction to measure the cycles of the program. We define the variable n to represent the height of the output image, and we compute the width with an aspect ratio of 16 : 9. Each benchmark session runs from $n = 100$ to $n = 2000$ in steps of 100. For each n we average four runs to get the cycle count.

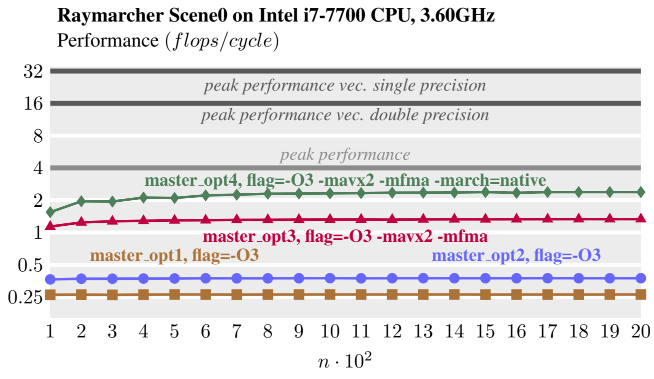
As an example, running a complete benchmark of *scene0* in our baseline code takes 4 to 6 hours of runtime, depending on the machine. For every phase, we provide data for the two scenes mentioned in Table 1. Measurements are taken on an **Intel i7-7700 CPU, 3.60 GHz** machine, with Turbo Boost disabled, which has a peak performance of 4 $flops/cycle$. So, for double and single precision SIMD vectorized code the theoretical peak performance is 16 and 32 $flops/cycle$.

Baseline. Table 5 show a constant performance for every n in the baseline. The peak percentage is very low as explained in section 3.

Scene	**Perf.** ($flops/cycle$)	**Percentage of peak**
scene0	0.173	1.087%
mandelbulb	0.540	3.375%

Table 5: Performance, which is constant for every n, and theoretical peak percentage of baseline, where peak performance is $16 flops/cycle$.

Optimization Phases. In Plot 6 and 7 performance and runtime of each optimization phase is provided. Note that we do not include the baseline results due to a FLOP count cut by almost 51% in scene0 and 14% in Mandelbulb.



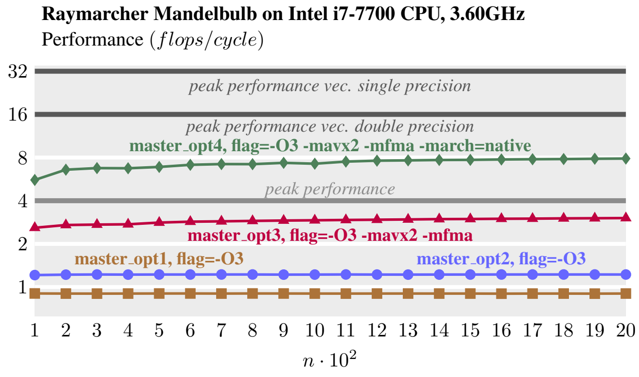
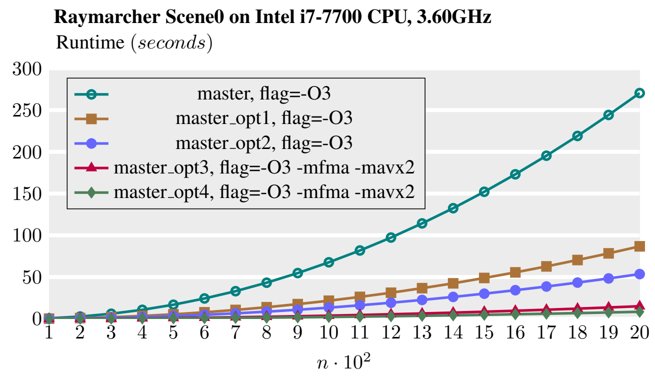


Fig. 6: Performance plots depicting the outcome of each optimization phase for both Scene0 and Mandelbulb using *GCC* 9.3.0.

Even with the significant FLOP reduction in master_opt1 for rendering scene0, we get a performance increase from 0.173 $flops/cycle$ to 0.264 $flops/cycle$. For the Mandelbulb scene, performance is almost doubled.

The improvement in master_opt1 is more noticeable in the runtime plots, where each of the scenes runtime is halved. Note that the Mandelbulb scene always has a higher performance due to its higher FLOP count. Moreover, there is no overhead in iterating over multiple shapes like in



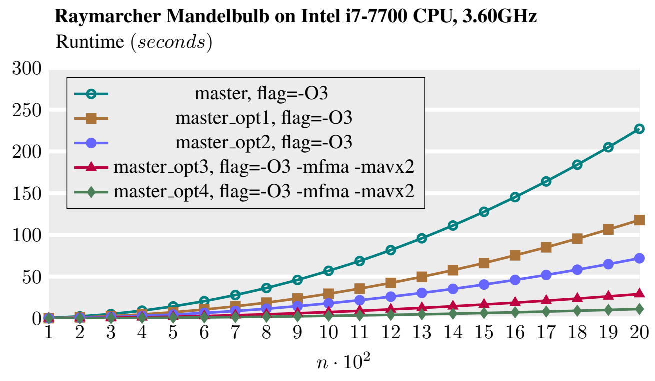


Fig. 7: Runtime plots depicting the outcome of each optimization phase for both Scene0 and Mandelbulb.

scene0, as explained in section 2.

In Table 6 the percentage of peak performance decreases from master_opt3 to master_opt4 for scene0, because of the higher theoretical peak. In Figure 6 depicted is the performance increase. Converting from doubles to floats increased performance by a factor of $1.8\times$ for scene0 and by $2.6\times$ for Mandelbulb.

In parallelization there is always some overhead, that makes itself more noticeable for smaller sizes and results in more speedup with higher sizes. This is why for smaller n, in our vectorized code, the performance gain is smaller than with higher n, and is not constant as before. The noticeable overhead for smaller n is due to the additional computation to keep track of each rays state via logic operations and mask computations that we didn't have before.

We also researched the performance on an AMD Ryzen 3700x processor and on the *ICC* compiler, which results in slightly better performance over *GCC*. However, for computationally intensive scenes we reach the best performance with *GCC*[10].

Scene	*SCENE0*		*MANDELBULB*	
Opt. Phase	**P**	**PP**	**P**	**PP**
master_opt1	0.264	1.65%	0.900	5.63%
master_opt2	0.375	2.34%	1.222	7.64%
master_opt3	1.332	8.33%	3.027	18.92%
master_opt4	2.383	7.45%	7.873	24.60%

Table 6: Performance and peak percentage for each phase of scene0 and mandelbulb for $n = 2000$. **P** is performance ($flops/cycle$) and **PP** is percentage of peak. The first 3 opt. phases have peak performance of 16 $flops/cycle$ and the last one has 32 $flops/cycle$.

5. CONCLUSIONS

We present an efficient approach to rendering images using the Ray Marching algorithm. Our method relies on certain assumptions about the scenes to avoid unnecessary computations. We combine loop unrolling, code motion, lightweight data structures, and other techniques learned in the Advanced Systems Lab lecture to make efficient use of the memory. In addition, we suggest approximations of high latency mathematical operations to lower computation-bound SDFs. Finally, vectorization of the entire rendering pipeline allows for a massive increase in performance by computing eight rays at a time. The resulting optimized rendering engine is considerably faster, up to $32.1\times$ speed-up from baseline for scene0. We show that the renderer is able to reach 7.45% of theoretical peak performance for a general scene containing multiple different shapes and up to 24.60% for a single complex shape scene. Although we considered only two scenes during performance analysis, we designed scene0 with a large variety of SDF shapes and materials to make sure that our optimization applies to a large spectrum of scenes. The main direction for further development, of the rendering engine, is multithreading and handling the resulting memory issues. Which is the key to a real-time Ray Marching rendered on the CPU.

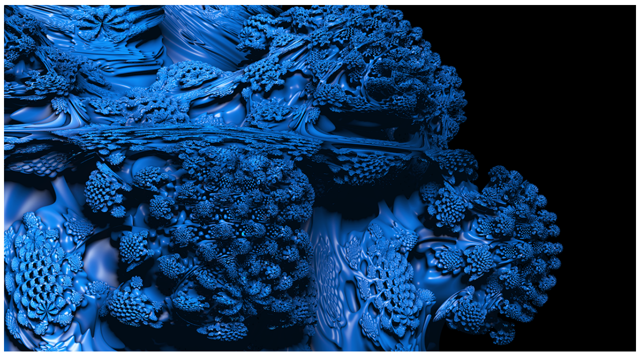


Fig. 8: Closeup of Mandelbulb scene.

6. CONTRIBUTIONS OF TEAM MEMBERS

Nihat.

- **Phase 1:** Worked with Alex on optimizing apply transform function, in particular moved computation of rotation values using sin, cos, and transformation to radians inside the scene creation section of our baseline.
- **Phase 2:** Worked on the *sdf* loop and *sdf* function. In particular tried loop unrolling the *sdf* loop with a factor two with two accumulators. Furthermore, applied code motion, precomputation, strength reduction, and inlined vector functions on *sdf cone* and *sdf box*. Refactored every function that takes scene as input to pass by reference.
- **Phase 3:** Worked on vectorizing most of the pipeline. In particular vectorized *render* loop, *trace*, *normal computation*, *specular coefficient computation*, *ray marching*, *sdf loop*, *sdf cone*, *sdf octahedron*, *sdf plane* and *sdf sphere* (see Section 3, Phase 3). Worked with Qais for shoot ray, implementing residual loop in *render* function and vectorizing recursive code for *reflection*. Worked with Alex and Qais to adapt log and exp approximations, also tried to fit polynomial using python.
- **Phase 4:** Tried finding appropriate hyper parameters (EPSILON, EPSILON_NORMAL) using random search and genetic algorithms on python for best output match between phase3 and phase4.

Qais.

- **Phase 1:** Focused on vector library optimization as described in Section 3. In particular, optimizing the vector library and the whole raymarcher codebase to use pointers and pass-by-reference and avoid the mentioned problems by having a more optimized layout.
- **Phase 2:** Worked again on the vector library. Changed the raymarcher from using *structs* as a datastructure to using pure C arrays. Moreover, optimized the memory handling of these arrays via pointers to not cause memory overhead. Then, worked on the optimization of the *sdf mandelbulb*, by doing loop-unrolling, simplifying expressions, code motion, and so on, which is described in paragraph SDF Mandelbulb in Section 3.
- **Phase 3:** Worked with Alex on Mandelbulb vectorization. In particular, implemented the masking infrastructure introduced in each for-loop to handle the individual returns of each ray depicted in Figure 5 and described in Section 3. Also, SIMD vectorized the *shoot_ray* method and the reflections in the raymarcher.
- **Phase 4:** Changed the whole raymarcher from using 64-bits floating points to using 32-bits floating points. Also made the change from shooting and processing four SIMD packed double-precision rays to eight SIMD packed single-precision rays.

Alex.

- **Phase 1:** Worked with Nihat on optimizing *apply transform* function, in particular added branching to avoid computing the function at every iteration if not necessary.
- **Phase 2:** Worked on *sdf* functions. In particular applied code motion and precomputation on *sdf* torus. Lowered branch misprediction on *sdf* octahedron. And minor changes to *sdf* mandelbulb. Removed branching in *sdf* cone. Worked with David on reducing amount of sqrt functions. Specifically, introduced the squared norm trick.
- **Phase 3:** Worked with Qais on Mandelbulb vectorization. Converted the base computation code depicted as c_i in Figure 5 to SIMD vectorized code. Worked on log and exp approximations more in detail trying to fit a polynomial using maple. Vectorized *sdf* box.
- **Phase 4:** Helped Nihat on finding right hyperparameters and bug fixes.

David.

- **Phase 1:** Implemented the *fast inverse square root* algorithm for single and double precision, the *newton method* approximation, and the SSE and AVX2 intrinsics for sqrt and also its reciprocal.
- **Phase 2:** Again tackled the square root bottleneck by reducing approximation errors. Worked with Alex on implementing the squared vector norm, avoiding square root calculations. Researched various compiler flags to analyze the effect on the square root.
- **Phase 3:** SIMD vectorized all the primitive vector operations for the representations shown in Figure 3. Also, SIMD vectorized the *sdf torus* and partially *sdf octahedron*. Worked with the icc compiler on Intel and AMD processors and analyzed the potential use Intel Intrinsics SVML instructions.
- **Phase 4:** Reduced unnecessary SIMD operations and fused multiplications and additions over the code.

7. REFERENCES

[1] Quilez Inigo, "Computer graphics blog - articles," https://iquilezles.org/www/index.htm, 1994 - 2020.

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

[3] Quilez Inigo, "Computer graphics blog," https://iquilezles.org/index.html, 1994 - 2020.

[4] Lourenço Bruno, "Cpu real time ray marcher," https://github.com/MadEqua/CPU-RTRM, 2019.

[5] team049, “Cpu-rtrm comparison,” https://gitlab.inf.ethz.ch/COURSE-ASL2020/team049/-/tree/master_opt5_rtrm_comparison/rtrm, 2020.

[6] David Eberly, “Fast inverse square root,” *Geometric Tools*, vol. 1, 2002.

[7] John D. Lipson, “Newton’s method: A great algebraic algorithm,” in *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, New York, NY, USA, 1976, SYMSAC ’76, p. 260–270, Association for Computing Machinery.

[8] Intel, “Intel intrinsics guide,” https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[9] José Fonseca, “José fonseca’s tech blog,” http://jrfonseca.blogspot.com/2008/09/fast-sse2-pow-tables-or-polynomials.html, 2008 - 2020.

[10] team049, “Performance comparison on intel and amd processors, gcc and icc compilers and three compilers flag combinations.,” https://gitlab.inf.ethz.ch/COURSE-ASL2020/team049/-/blob/master/plot/plot6.png, 2020.