15-418 Project Proposal: Parallel Ray Tracer

Samuel Schaub and Nader Daruvala

November 16, 2023

1 URL

https://ndaruvala.github.io/15418-final-project/

2 Summary

We aim to implement a parallel ray tracer that utilizes SIMD instructions, ray packeting, ray reordering, and global ray reordering techniques to achieve a significant speedup over a sequential implementation.

3 Background

Introduction: Ray tracing is a technique for modeling the behavior of light in a scene of objects. This technique is used to render photo-realistic images. Ray tracing can simulate reflections, shadows, blur, and refraction accurately. Ray tracing has uses in many applications, such as creating architectural renders, lighting designs, video game graphics, and animations.

Technique: In this technique, rays of light are extended from the camera (an imaginary human eye), and their paths are traced as they intersect with and reflect off of objects toward sources of light. The algorithm calculates the color of each pixel by determining the amount of light projected onto the pixel by every ray that intersects with it and taking into account the specific qualities of the material at that pixel.

Bounding Volume Hierarchy (BVH) Tree: In any given scene, there can be anywhere from a few hundred thousand to a few billion geometric primitives. Naively testing for ray intersection with each of these primitives would turn out to be too costly. To do fast intersection tests with rays, you can utilize a bounding volume hierarchy (BVH) which is a tree data structure for geometric objects. The BVH tree partitions the 3-D space into nodes that specify bounding boxes of their respective partition of space. Each leaf of the tree represents a bounding volume that contains a handful (4 - 10) of primitives. BVH allows us to search the space in logarithmic time in the number of primitives.

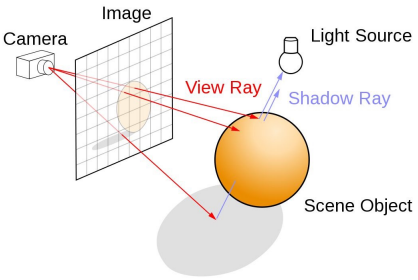

Figure 1: Ray Tracing Technique (developer.nvidia.com/discover/ray-tracing)

Algorithm: The following pseudocode describes the high-level algorithm for tracing ray intersections with geometric primitives using a BVH tree. It is a modified version of the code on slide 20 from CMU's 15-462 lecture 8 slides.

```
void hit(Ray* ray, BVHNode* node, HitInfo* best) {
 // test if ray hits node's bbox
 HitInfo hit = intersect(ray, node->bbox);
 if (ray does not intersect with node's bbox) return;

 // for leaves, check each primitive
 if (node is a leaf) {
    for (each primitive p in the node) {
        hit = intersect(ray, p);
        if (ray intersects with p and p is closest primitive so far) {
            best.prim = p;
            best.t = t;
        }
    }
 } else {
    // traverse BOTH children
    hit(ray, node->child1, best);
    hit(ray, node->child2, best);
 }
}
```

4 The Challenge

When doing ray tracing, ideally there are a large number of rays to get a good quality image. To speed up the process, we need a high throughput of rays, however, there are many challenges to increase this throughput. Each ray in the ray tracer varies a lot in the time to compute because each ray has a completely different path than the others and bounces a different number of times. Additionally, each ray will hit different objects and different materials which means that ray tracing inherently has a lot of conditional execution. This means that the two biggest challenges in creating a parallel ray tracer will be the workload balance and the divergent execution. Furthermore, another aspect that will make increasing the ray throughput challenging is that data access will be very slow due to the lack of locality. The ray tracer access to the BVH is frequent and

unpredictable because each ray will be accessing different parts of the tree and thus suffers from poor cache performance and is bottlenecked by bandwidth.

5 Resources

We will utilize the slides from 15-469, and the following research papers: Adaptive Ray Packet Reordering (Boulos), Understanding the Efficiency of Ray Traversal on GPUs (Alia et. al), Ray tracing and volume rendering large molecular data on multi-core and many-core architectures (Navratil et. al), Architecture Considerations for Tracing Incoherent Rays (Alia).

Additionally, for starter code, we are going to ask Ioannis Gkioulekas, the professor of 15-468, for the starter code of that class. If he agrees then we will use that for starter code and if not we will have to either do it from scratch or find different starter code. To develop the code, we will be using our laptops which have the Apple M1 Pro and M2 Pro processors, each with 16GB of RAM.

6 Goals and Deliverables

6.1 Plan to Achieve

We plan to achieve a functioning implementation that uses ray packeting with ray reordering. Ray packeting is the grouping of rays into a vector. The rendering computation on each ray in the group will be executed in parallel using vector instructions. However, this causes a lot of code divergence because some rays in the vector will intersect a given BVH node leaf while other rays will not, leading to low SIMD utilization and wasted resources. To solve this issue, we will implement ”ray reordering” which will regroup ”active” rays (rays that all intersect at a given BHV node) together into vectors when the packet utilization drops below a certain threshold.

6.2 Hope to Achieve

We hope to achieve global ray reordering to optimize the BVH for L1 and L2 cache. Global ray reordering is the division of the BVH tree into subtrees, each of which is sized for the L1 and L2 cache. Each of the subtrees have a work queue, and when a packet enters that subtree then we add it to its queue. When the subtree’s queue is sufficiently large, intersect the enqueued packets with the subtree which amoritzes the treelet load over teh enqued packets. As a stretch goal, we can also adapt our implementation to use ISPC tasks and execute on multiple cores in parallel.

7 Platform of Choice

To develop the code, we will be using our laptops which have the Apple M1 Pro and M2 Pro processors, each with 16GB of RAM. We will test our implementation on both the gates clusters and our machines. To write SIMD code, we will be using the ISPC compiler as we did in assignment 1. We will be writing both C and C++ code. Using ISPC will make it easier to develop SIMD code by using the SPMD abstraction. This will be much more productive than writing SIMD intrinsics by hand.

8 Schedule

8.1 Week 1

Implementing all code necessary for scenes and rasterization, and also implementing sequential ray tracing.

8.2 Week 2

Implementing ray packeting with SIMD instructions.

8.3 Week 3

Implementing ray reordering to increase SIMD utilization and reduce divergent execution.

8.4 Week 4

Implement global ray reordering to optimize BVH for L1 and L2 cache.