Parallel Ray Tracing

15-418 Final Project

Nader Daruvala
Carnegie Mellon University
Pittsburgh, PA
ndaruval@andrew.cmu.edu

Samuel Schaub
Carnegie Mellon University
Pittsburgh, PA
sschaub@andrew.cmu.edu

I. Summary

We implemented a parallel ray tracer in the ISPC language on the CPU. We implemented various different strategies to obtain significant speedup over the sequential version of the algorithm. Github

II. Background

A. Motivation

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.

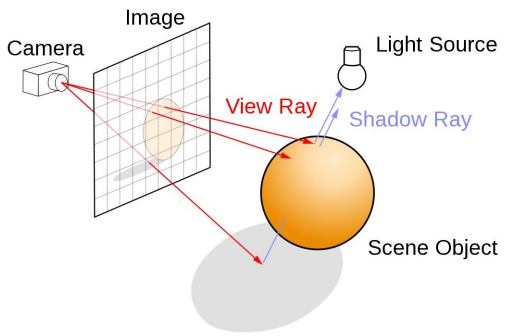

Fig. 1. Ray Tracing Diagram (developer.nvidia.com/discover/ray-tracing)

B. 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.

C. Problem

The input to the algorithm will be a list of objects (primitives) in the scene and their locations in the 3D plane. In addition, we will need the camera information. This will include the position of the camera, its viewing angle, the width of the image in pixels, the aspect ratio of the image (ratio of image width to image height), the number of rays to trace per pixel in the image (samples per pixel), and the maximum number of times a pixel can bounce. The output of the image will be a matrix of integers ranging from 0-255, representing the RGB values at each pixel in the output image.

Each pixel value will be the result of averaging the color accumulated by all randomly generated sample rays within the half-unit square centered around that pixel point. In the project, all these RGB values will be stored in the form of a Portable Pixmap Format (PPM) image, which encodes the resolution of the image and all the aforementioned RGB values.

D. Key Data Structures

The most obvious approach to solving this problem would be to iterate over each pixel, generate all the rays in that pixel, and trace each one sequentially by intersecting each ray with every single primitive given to find the closest object that ray hits and bounces from. After each bounce, the ray will accumulate some amount of color from the objects it intersected. The idea would be to trace each ray till it either escapes the scene or reaches the maximum recursion depth specificied by the input.

However, note that 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 far too costly. To do fast intersection tests with rays, we 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 (8 - 16) of primitives. BVH allows us to search the primitive space in logarithmic time in the instead of linear time. Some of the other key data structures used in the code are the “Ray” datastructure and the “HitRecord” datastructure. A ray contains an origin point and a direction vector. A hitrecord records various important pieces of information relating to a ray intersecting with a primitive or BVH node, including the intersection time, the type of material hit, the intersection point, and the normal vector to

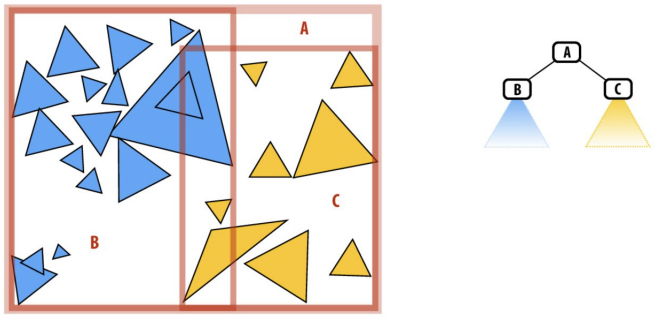

Fig. 2. BVH Tree Diagram (CMU 15-462 Slides)

the intersection point (which is useful for calculating the new ray that bounces off the material).

E. Algorithm

The pseudocode shown in "Algorithm 1" 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.

Algorithm 1 Key Operation: Hit Function on BVHNode

```
1:  procedure HIT(Ray* ray, BVHNode* node, HitRecord* best)
2:      HitRecord hit ← intersect(ray, node→bbox)
3:      if ray does not intersect with node's bbox then
4:          return
5:      end if
6:      if node is a leaf then
7:          for each primitive p in the node do
8:              hit ← intersect(ray, p)
9:              if ray intersects with p and p is closest primitive so far then
10:                 best→prim ← p
11:                 best→t ← t
12:             end if
13:         end for
14:     else
15:         HIT(ray, node→child1, best)
16:         HIT(ray, node→child2, best)
17:     end if
18: end procedure
```

F. Workload and Opportunities for Parallelism

The most computationally expensive part of the algorithm is the cost of intersecting a ray with the BVH tree or, in the naive approach, iterating over all primitives to find the closest one that was hit by the ray. One of the major bottlenecks in a recursive or "depth-first" style of ray tracing as shown above is that each ray has virtually a random path, making it very different from any other ray traced prior. These frequent and unpredictable accesses to the BVH leads to very poor cache locality because each ray has to intersect a seemingly random set of primitives. When the scenes are very complex, this issue is amplified, leading to a massive memory bandwidth bottleneck. Due to the massively data independent nature of

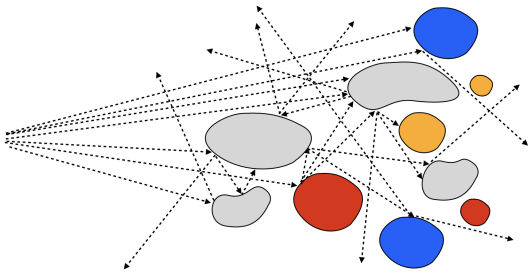

Fig. 3. Ray Incoherence Visualized (CMU 15-869 Slides)

ray tracing, there are a many opportunities for parallelism. Firstly, we can tile the image up and process each tile of the image on a separate CPU core. For each tile, we can also parallelize over the pixels or parallelize over the rays since each pixel is independent from other pixels and each ray can be traced separately without affecting the result of any other rays. Within a pixel, there are also little to no dependencies. The only dependency is that the colors of all the rays sampled within a pixel have to be added together and divided by the samples per pixel value to calculate the final RGB value for that pixel. This is a very weak constraint that can be solved by reducing our ray color values after tracing all the rays in a pixel to completion. Using the SIMD execution model is a great fit for this problem as it is inherently very data-parallel.

III. Approach

A. C++ and ISPC

In our project, we divided tasks between C++ and ISPC (Intel SPMD Program Compiler). C++ was primarily used for setting up scene information and taking in user input. This included initializing the 3D environment, defining objects and their locations, lights, camera parameters, and preparing the data structures necessary for ray tracing and image output. Once the scene was set up, the output image memory was passed over to ISPC. ISPC's role was to perform the computationally expensive tasks of ray tracing including rendering and writing results back to the output image array. We decided to use ISPC to execute SIMD operations, which are central to doing ray packeting which we will discuss later. For our project, we used the Apple M2 Pro chip which utilizes NEON intrinsics which is the SIMD architecture for ARM processors.

We compiled our code to i32x8 intrinsics, which are able to operate on 8 32-bit values in parallel. The M2 Pro chip consists of a total of 12 cores, 8 of which are high-performance cores and 4 of which are high-efficiency cores. In our testing, using all 12 cores provided diminishing returns compared to using only the 8 high performance cores, so we used a maximum of 8 cores to keep our results more consistent.

For the initial sequential version of our code, we followed through the book Ray Tracing in One Weekend by Peter Shirley.

B. Parallelizing over Pixels

Asssuming a SIMD target size of 8, meaning we are able to process 8 32-bit floats in one vector instruction, one can then process 8 pixels at a time in parallel. Each pixel will be mapped to an ISPC program instance, which will then iterate over all of the ray samples in its pixel and average the total color accumulated to obtain the resulting pixel value. This method provides a roughly 5x speedup. This is the most obvious approach and can provide a good enough implementation if one is not looking for greater speedup. However, there is not much upside potential to utilizing the cache more efficiently because each ray is still being processed separately and thus accessing the hittable objects in the scene unpredictably and frequently. Additionally, this results in a large amount of divergence of instruction streams due to each ray hitting different objects with different materials. This leads to low SIMD utilization which we will try to solve later.

C. Tiling

So far, we have only been utilizing SIMD capabilities, which are only used within a single core. We would like to utilize all of our 8 processing cores, which are so far sitting un-utilized. So, the next most obvious optimization is to tile the image into sections where each processor renders its own tile in parallel. Since this project utilized ISPC, we utilized ISPC tasks to assign each processor its own task, which will execute SIMD operations. This addition gave us a roughly 10-14x speedup over the purely sequential implementation with no tiling. Moving forward, we will utilize this method in all our next iterations of our algorithm because it provides a great speedup with no downside.

D. Aray of Structures (AoS) vs Structure of Arrays (SoA)

One major issue we noticed when trying to optimize ray tracing with SIMD is the poor performance of reading and writing to memory using the usual data format we would normally use when writing serial C/C++ code. This is the ”array of structures” (AOS) layout. For example, we originally had a single array that represented our image where each element had a structure containing red, green, and blue integer values. However, we found that this lead to poor cache performance with SIMD code. This is because when the data is laid out this way, the vecotrized instructions are loading data that is farther apart which decreases locality and thus performance. As shown, even if the program instances are accessing the elements of continuous structures, the actual address of the data is far apart which hurts performance because there is a lack of cache locality and it requires the SIMD vectors to do a gather. This is because when indexing into an array with SIMD, each program instance will have to scatter and gather depending on reading or writing to the other instances to ensure correctness. To solve this problem

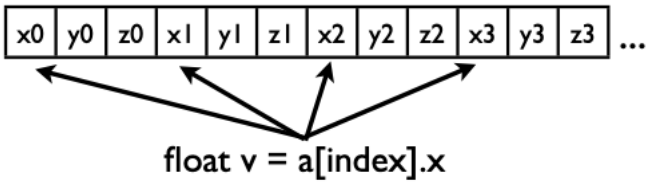

Fig. 4. AOS Diagram (ISPC Intel 2011)

of AOS for SIMD instructions, you can change the layout of the data to be a ”structure of ararys” (SOA). In our case, that meant we separated red, green, and blue into their own arrays, each of which are sized for the target SIMD width. Then, we put those arrays into a structure. This allows contiguous data to be loaded and stored with no scatters and gathers in a single SIMD instruction. This optimization of the structure

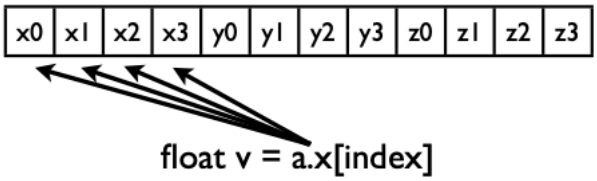

Fig. 5. SOA Diagram (ISPC Intel 2011)

of the data removed all the unnecessary gathers and scatters and increased the cache locality leading to an increase in performance. Our results found a 2x speedup when switching from AOS to SOA data layouts while keeping vector width and number of tasks the same. The transformation from AOS to SOA is especially advantageous in SIMD because it aligns well with how SIMD processors operate. SIMD processors are designed to perform the same operation on multiple data points in parallel. When the data is stored in an SOA format, it becomes easier for these processors to load and operate on homogeneous data efficiently. In addition to improving SIMD efficiency, SOA also enhances cache coherency. The SOA format ensures that data elements which are processed together are also stored closely in memory. This significantly reduces cache misses. In ray tracing, where calculations are often repetitive and continuous over similar sets of data (like pixel colors), having an efficient cache usage pattern is crucial.

E. Cache Optimized BVH Layout

In our ray tracing project, a critical optimization was implemented in the structure of the BVH. Originally, the BVH was constructed using pointers to navigate between left and right child nodes. While conceptually straightforward and easy to implement, this pointer-based approach resulted in suboptimal cache performance. This was due to the non-contiguous memory allocation of nodes, which led to frequent cache misses during the traversal process. To address this, we transitioned to an array-based representation of the BVH nodes. In this

structure, all nodes of the BVH are stored in a contiguous array. Each node in this array contains necessary information such as bounding box data and indices pointing to its children in the same array. This shift to a sequential memory layout significantly improved cache locality, as contiguous memory accesses are more cache-friendly.

Another crucial aspect of our optimization involved the way hittable objects are stored and referenced within the BVH. Instead of having a scattered set of objects, we organized them into a contiguous array. This array was then sorted in a manner that aligns with the structure of the BVH. Specifically, each leaf node of the BVH references a contiguous subarray of hittable objects. This contiguous storage means that when a ray intersects a leaf node of the BVH, all potentially hittable objects within that node are located in a continuous memory block. This arrangement greatly enhances data locality, reducing cache misses during the ray-object intersection tests, which are among the most performance-critical parts of ray tracing. The combination of an array-based BVH and contiguous storage of hittable objects had a profound impact on the performance of our ray tracing algorithm. It significantly reduced the memory overhead associated with pointer-based node structures and minimized cache misses during BVH traversal and intersection tests. By implementing this cache optimized BVH layout, we obtained a further 2x speedup over our previous parallel implementation with a non optimized BVH layout.

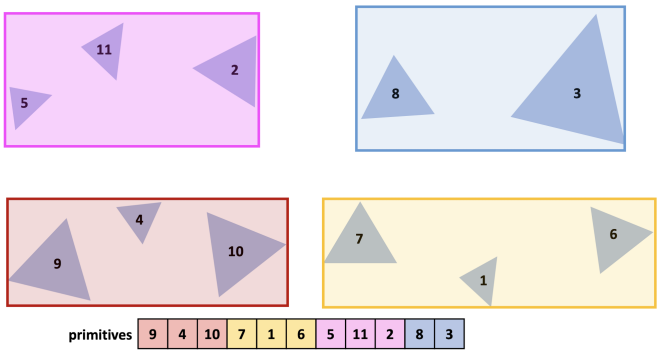

Fig. 6. BVH Memory Layout (CMU 15-462 Slides)

F. Ray Packeting

One of the most significant bottlenecks we encountered in our ray tracing optimization efforts was the issue of ray incoherence. In ray tracing, rays often bounce off surfaces in random directions. This randomness, or incoherence, of rays leads to a divergence in the instruction streams when processed using SIMD. As a result, the SIMD units, which are most efficient when executing the same operation on multiple data points, suffer from low utilization. This inefficiency is particularly pronounced when vectorizing over individual pixels, as each pixel's ray might follow a drastically different path.

To mitigate this challenge, we shifted our approach from vectorizing over individual pixels to vectorizing over groups of rays, termed "ray packets." A ray packet is a collection of rays that are processed together as a unit. By grouping rays into packets, we could exert more control over the scheduling of rays, aligning them to be more coherent in their paths and interactions with the scene. This method allows us to maintain higher SIMD utilization, as the rays within a packet are more likely to follow similar paths and undergo similar interactions with the scene. Therefore, the SIMD units can operate more effectively, executing parallel instructions on these coherent ray packets. The use of ray packets also had a positive impact on cache performance. By tracing packets

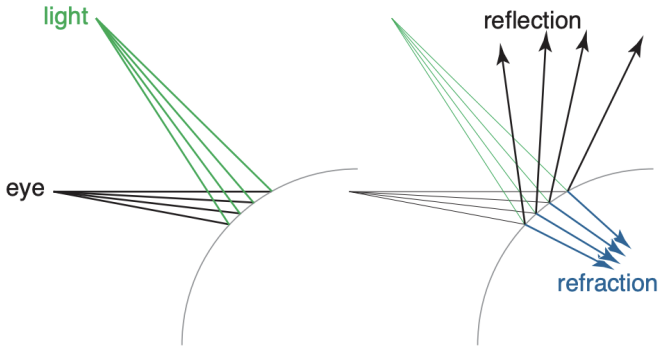

Fig. 7. Packeting Coherent Rays (Boulos Et. all, 2007: Packet-based whitted and distribution ray tracing)

of coherent rays, the likelihood of accessing similar data or nearby data in memory increases. This similarity in memory access patterns leads to improved cache hit rates, as the data required for processing one ray in the packet is more likely to be beneficial for processing other rays in the same packet. Additionally, the ray packet approach allows for more sophisticated scheduling policies. We were able to implement strategies that prioritize tracing rays with similar directions or that are likely to intersect with the same objects. This further enhanced the coherence of the rays within each packet, optimizing both cache performance and SIMD utilization.

G. Global Ray Queueing

As our final iteration over the implementation, we introduced a technique called global ray queuing. This method further refines our approach to handling rays within the BVH structure, focusing on enhancing cache efficiency and SIMD utilization. Traditionally, when tracing a packet of rays, each ray is followed through the BVH down to the leaf nodes where it intersects with primitives. However, this approach can still lead to inefficiencies, because the primitives are constantly being loaded in and out of the cache for each packet that is traced. Global ray queuing addresses this by accumulating rays that intersect a specific BVH leaf node into a queue. Instead of immediately processing these rays, we defer their intersection tests. Once a substantial number of rays have been queued for a particular leaf node, we release this queue for processing.

At this point, we vectorize over the entire set of queued rays, performing intersection tests with the primitives in the leaf node. This ensures that all rays being processed are likely

to interact with the same set of objects within the leaf. The key advantage of global ray queuing is its ability to maximize the utilization of the primitives stored in the cache. Since all rays in a queue are likely to intersect with objects in the same leaf node, the required data (such as vertex positions, normals, and material properties of the primitives) remains in the cache throughout the processing of the queue. This results in significantly reduced cache misses. Furthermore, this technique also addresses the issue of ray incoherence. By processing rays that are likely to intersect with the same set of primitives, we ensure a higher degree of coherence among the rays in the queue. This coherence directly translates to improved SIMD utilization, as the SIMD units can now operate more effectively on a set of rays with similar paths and intersection characteristics.

Implementing global ray queuing required us to rethink our ray tracing pipeline, particularly in how we manage and schedule ray intersections within the BVH. The approach adds a layer of complexity, as it involves dynamic management of ray queues and determining the optimal point at which to process these queues. However, the performance gains in terms of cache efficiency and SIMD utilization have been substantial, making this complexity worthwhile.

IV. Results

In this project, we evaluated the impact of various bounding volume hierarchy (BVH) optimizations on ray tracing performance. Our tests were categorized into four groups: no BVH, naive BVH, cache-optimized BVH, and global ray reordering. We also varied vectorizing over pixels vs ray packeting. The goal was to assess how each method affects rendering efficiency in scenes of varying complexity. We conducted tests on three types of scenes: large-scale scenes with nearly 1600 objects, medium sized scenes with 400 objects and smaller scenes like the Cornell Box with 8 objects. Render time was used to evaluate each BVH method's effectiveness.

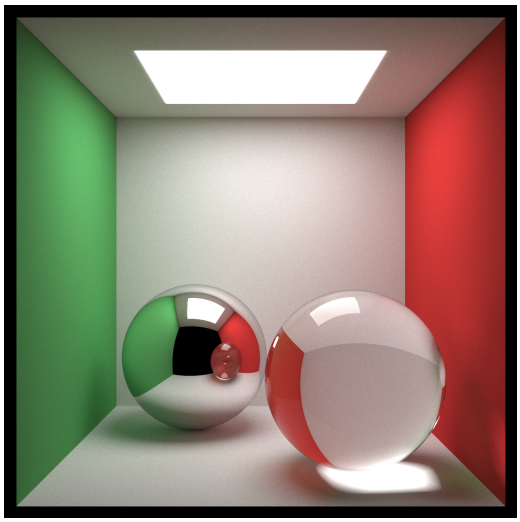

Fig. 8. Cornell box with mirror and glass spheres, 1280x1280, 2048 samples per pixel

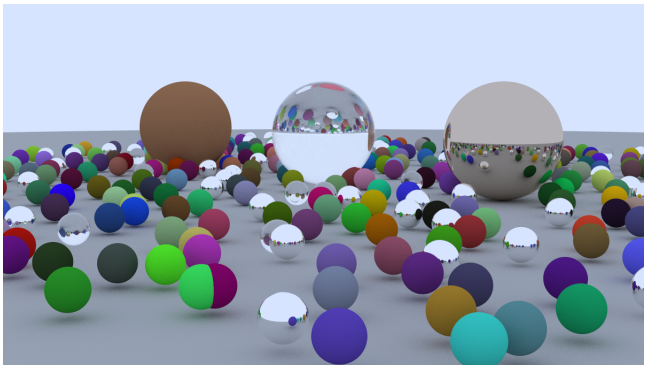

Fig. 9. 400 spheres varying materials, 1200x675, 200 samples per pixel

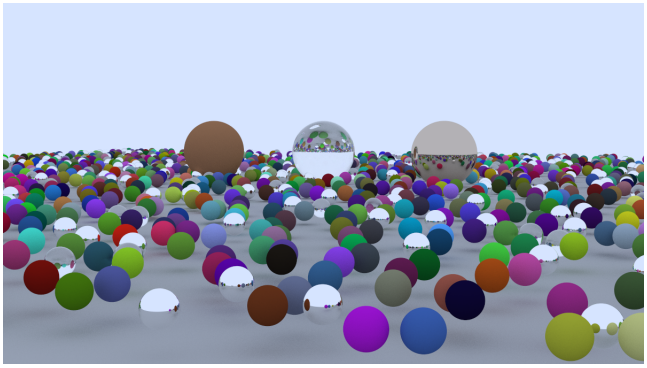

Fig. 10. 1600 spheres varying materials, 1200x675, 400 samples per pixel

1) Cornell Box Scene Analysis: The Cornell Box scene, consisting of a few objects with complex lighting, was rendered at 400x400 resolution with 200 samples per pixel. The smallest of our test scenes, it revealed the following:

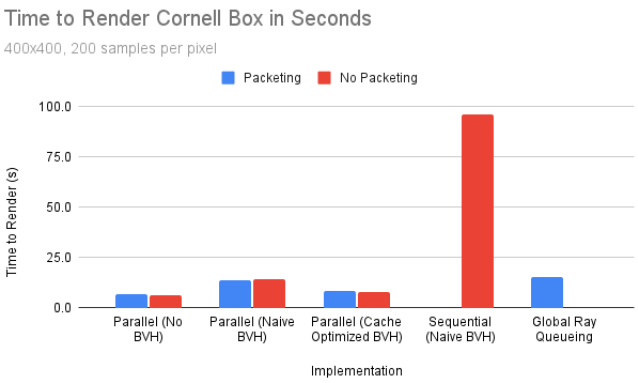

Fig. 11. Cornell Box Results

Global Ray Queuing was the least efficient method among the parallel methods, likely due to the overhead of queuing and keeping track of ray states not being compensated by the benefits of fast primitive searching in very simple scenes with only a few primitives. Although the Parallel method with a cache optimized BVH outperformed the Naive BVH, it was

still not faster than the Parallel without BVH for the same reasons above. As a result, we can see that the building a BVH is not worth it when rendering an extremely simple scene with only a few primitives, such as in the Cornell Box scene. Nevertheless, Cache Optimized BVH closely followed behind No BVH in speedup, showing the power of cache locality. The best speedup obtained over the sequential implementation was 15x by the No BVH method.

2) 400 Random Spheres Scene Evaluation: In a more demanding scene with 400 random spheres at a resolution of 1200x675 and 64 samples per pixel, we observed that:

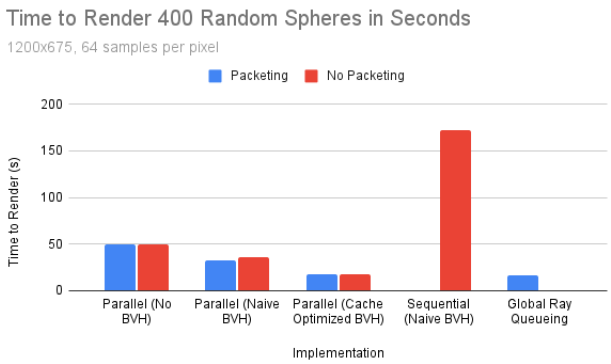

Fig. 12. 400 spheres Results

Global Ray Queuing slightly outperformed all other methods, suggesting that the high overhead barely pays off even for medium sized scenes. Cache Optimized BVH was still effective, though less so than global ray queuing, reinforcing the importance of optimized memory access. No BVH and Naive BVH showed comparable results, with the latter offering marginal improvements.

3) 1,600 Random Spheres Scene Results: Our most complex scene, with 1,600 random spheres, highlighted the strengths and limitations of our optimization techniques:

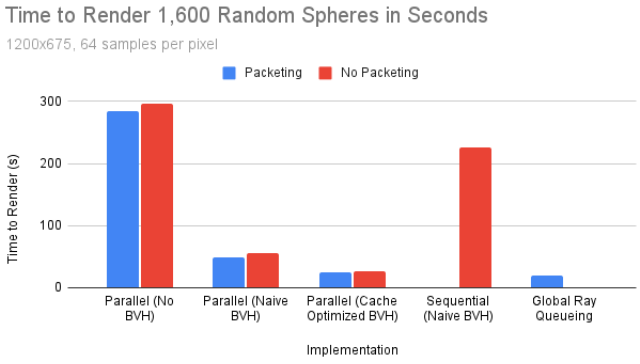

Fig. 13. 1600 spheres Results

Global Ray Queuing continued to excel, delivering the fastest rendering times, and showcasing its potential for handling high-complexity scenes. Cache Optimized BVH remained robust but was outclassed by global ray queuing, emphasizing the benefits of our final optimization in large-scale scenes. No BVH performed extremely poorly, showing that the benefits of having a BVH when there is a large set of primitves to search through really pays off, even moreso than the benefits of paralleism (as shown by the fact that sequential with naive BVH performed better than parallel with no bvh).

The results highlight a clear trend: the effectiveness of each BVH optimization is highly dependent on the complexity of the scene. In large-scale scenes, global ray reordering excels by efficiently handling ray incoherence and maximizing SIMD utilization. However, its performance advantage diminishes in smaller scenes where the overhead becomes more pronounced than the benefits. Conversely, the cache-optimized BVH shows a consistent performance improvement in both large and small scenes, making it a versatile choice for various scenarios. The naive BVH, while better than no BVH in complex scenes, demonstrates the importance of sophisticated optimization techniques to truly harness the potential of BVH in ray tracing. These results underscore the importance of choosing the appropriate BVH optimization technique based on scene complexity. While global ray reordering offers significant advantages in large, complex scenes, its overhead makes it less suitable for simpler scenes where cache-optimized BVH proves to be more effective. Understanding these nuances is crucial for optimizing ray tracing performance across different types of graphical environments.

In our larger test scenes, we found that the rendering times were significantly improved by optimizing cache usage. However, the performance was still not as high as expected, hinting at potential memory-bound limitations. Further investigation into memory access patterns could reveal whether the system is indeed memory-bound or if the bus transfer rates are a bottleneck.

SIMD divergence was another area of concern. Despite our efforts to group coherent rays via packeting, there was still a notable variance in SIMD utilization that was unavoidable. This is particularly true in complex scenes where rays can diverge after interacting with different objects. The performance data from the cache-optimized BVH implementation versus global ray queuing indicates that while the latter improved SIMD utilization, there was still room for improvement.

REFERENCES

[1] G. Ray Tracing in One Weekend. raytracing.github.io/books/RayTracingInOneWeekend.html Accessed 15 12. 2023.

[2] Ray Tracing: The Next Week. raytracing.github.io/books/RayTracingTheNextWeek.html Accessed 15 12. 2023.

[3] Aila, Timo and Tero Karras. “Architecture considerations for tracing incoherent rays.” High Performance Graphics (2010).

[4] M. Pharr and W. R. Mark, ”ispc: A SPMD compiler for high-performance CPU programming,” 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 2012, pp. 1-13, doi: 10.1109/InPar.2012.6339601.

[5] Boulos Et. all. ”Packet-based whitted and distribution ray tracing.” Proceedings of Graphics Interface 2007.