15-400 Project Report

Max Slater (Advisors: Rohan Sawhney, Keenan Crane)
mjslater

---

1 Introduction

1.1 The Problem

Closest point queries, or CPQs, solve a straightforward problem: relative to an input location, where is the closest point on some surface? When working with, for example, triangle meshes, this is easy to compute by simply taking the minimum distance to each triangle. However, we can use various acceleration structures to compute closest points far faster than linearly in the input size.

CPQs are a foundational tool in geometric computing, as many relevant algorithms depend on them, including but not limited to collision detection and simulation, simulating distance-dependent forces, computing signed distance fields, sphere tracing implicit surfaces, and Monte Carlo geometry processing. All of these algorithms use CPQs as a basic primitive, so their performance depends heavily on the efficiency of computing millions or billions of CPQs. If we can produce a faster implementation of CPQs, the performance of many other geometric algorithms stand to benefit.

1.2 Approach

Our goal was to determine what kind of algorithmic and systems-level tradeoffs are desirable for efficiently computing closest point queries. We build on previous work in high performance ray tracing, exploring which optimizations useful for ray intersections carry over to closest points. To determine the best options, we implemented various types of bounding volume hierarchies, or BVHs, on both the CPU and GPU. We have produced a vectorized, parallel CPU library for closest points, as well as a parallel GPU compute implementation, and our benchmark results are reported here.

1.3 Related Work

Much previous work on high performance ray tracing, a very similar problem, has focused on improving the construction quality and query performance of BVHs. A BVH is a tree structure where each sub-tree represents a subset of primitive shapes and contains a bounding volume enclosing this set. A BVH can then be used to accelerate finding ray intersections or closest points, as in both cases we may define a recursive traversal rule. Our rule describes when to traverse an interior node's child: only if it stands to improve the best result seen so far.

However, there are many different ways to construct and query a BVH. In order to get the best acceleration structure for a given problem, we can choose from many parameters, such as the following:

1. Branching factor
2. Leaf size
3. Node splitting heuristic (e.g. surface area, volume, overlap, etc.)

4. Node bounding volume (e.g. axis aligned boxes, oriented boxes, etc.)

Additionally, there are several ways to take advantage of parallelism. Once a suitable BVH is built, the simplest strategy is to parallelize over many independent queries. Since the results of different queries do not depend on each other and do not mutate the BVH, we can straightforwardly partition our input into various sets to be executed on different processors.

However, applying SIMD (single instruction, multiple data) parallelism is less straightforward: we may vectorize independent queries, but previous work on ray tracing has indicated that execution divergence during BVH traversal may become a significant issue. Hence, it is also possible to vectorize the BVH itself. If one builds a tree with a branching factor greater than 2 (e.g. 4, 8, or 16), the children of a node may be packed into SIMD vectors and checked in parallel during traversal for a single query.

The primary computational workload of all BVH-based algorithms is the tree traversal, with extra numerical computations performed at the leaves of the BVH. This is a difficult task to optimize, as we must consider all of the following factors:

1. Tradeoff of traversal cost and per-primitive numerical computation
2. Memory access dependencies between tree levels
3. Locality of node accesses
4. In the SIMD case, execution coherence
5. Cost of recursive or stack-based iterative approaches

1.4 Contributions

Broadly, we find that despite their similarities, computing closest points is a much more computationally expensive task than ray intersections. CPQs typically have to traverse a large subset of nodes within a BVH, particularly near the medial axis. Query vectorization strategies typically struggle with divergence, even for coherent inputs.

On the CPU side, our work confirms that most previous techniques for high performance ray tracing also hold true for CPQs. For example, we saw significant gains using vectorized trees. However, not seen with rays, we found that using more complex bounding volumes, such as oriented bounding boxes, often increases performance due to improved primitive culling.

On the GPU, we find that no BVH-based approach gives orders of magnitude improvement over the CPU implementation, as all struggle with execution divergence. Such divergence is particularly harmful to performance in the presence of poor culling, as every query in a vector lane must check many disparate primitives. However, we note that some GPU approaches are viable, focusing on the use of both stack-less BVH traversals and wide traversals with ordering optimizations.

2 Approach

2.1 CPU

Our CPU implementation is a custom C++ codebase using the Enoki library for vectorization. It supports building BVHs with various splitting heuristics, bounding volumes, and SIMD vectorization widths.

On the CPU, queries are performed via standard recursive BVH traversal. We support both thread-based and SIMD parallelism, but through different approaches. Thread based parallelism simply divides the input queries into blocks that are scheduled onto various worker threads in a thread pool. SIMD parallelism is introduced by widening individual BVH nodes: we take a normal binary BVH and collapse the children of interior nodes into that node. This process can be repeated N times to get a branching factor of 2^N. The resulting BVH nodes contain a SIMD-width collection of child bounding boxes and node indices. Additionally, contiguous primitives within a leaf node are packed into lane-width packets.

The following options are supported:

1. Build heuristics: longest-axis center, surface area, volume, overlap surface area, overlap volume
2. Node bounding volume: axis aligned box, sphere, oriented box, sphere-swept rectangle
3. Threads: arbitrary number
4. SIMD width: 1, 4, or 8

2.2 GPU

Our GPU implementation is also custom C++ codebase. It uses Vulkan to interface with the GPU, where various BVH traversals are implemented as compute shaders. BVH construction is performed on the CPU, before being packed into GPU-resident buffers and sent to compute shaders.

All BVH traversals follow the same basic formula. The BVH is represented as a pair of contiguous GPU buffers, one holding a linearized tree of BVH nodes, and one holding a sorted list of primitive shapes. The node array is arranged via a depth-first traversal, in order to match the spatial access behavior of most traversals. BVH nodes contain a volume bounding all primitives in its subtree. Interior nodes reference the index of exactly two child nodes, and leaf nodes refer to a contiguous region of the primitives array.

All approaches are invoked in a 1D compute dispatch of $n \times 1 \times 1$ size, where n is the number of queries. Block size is not specified when using compute shaders, as this is decided by the driver. Queries are batched into 32-wide segments, which are executed by individual warps. No cross-warp shared memory is required. Lastly, our benchmarking code can generate both spatially contiguous, as well as random-order, queries in order to test the effects of input coherence.

The GPU implementation supports the following traversal strategies:

1. Explicit stack-based

   The traversal shader declares an explicit stack to emulate a recursive traversal. This is simply an array of lane-local integers. When a node is evaluated, its children are pruned if possible, and pushed onto the stack in closest-distance order.

2. Explicit stack-less

   Introduced in the paper *Efficient Stack-less BVH Traversal for Ray Tracing*, this approach eschews the implicit stack, instead only storing a single traversal state marker representing whether the traversal should continue to the left child, right child, or parent. This means we do not need the array of integers, but it also prohibits us from 'enqueueing' closer children before farther.

3. Threaded stack-less

   This approach also does not require an explicit stack, instead opting to add a member to each node that specifies where to traverse to in the event the node is culled.

4. Wide BVH

   This approach increases the branching factor of the BVH to an arbitrary power of 2. This reduces the depth of the tree, but requires more children to be checked at each node. For branching factors higher than 2, an explicit stack-based traversal is required. Widening the tree also allows us to choose how to prioritize traversing children. This can be in arbitrary order, closest-first, or sorted by distance.

5. Oriented Bounding Box BVH

   This is not a specific traversal approach, but rather a BVH where nodes' bounding volumes are represented as oriented bounding boxes (as opposed to axis-aligned boxes). When constructing the BVH, we compute an optimal primitive partitioning based on the same type of heuristics as the axis-aligned case, but using oriented boxes fitted to primitives using principal component analysis.

   Our OBB tree may be traversed in both an explicit stack based, and explicit stack-less manner.

3 Evaluation

3.1 CPU

3.1.1 Threading/SIMD

CPU benchmarks were run on an 18-core, 36-thread Intel Core i9-9980XE running at ~ 3.8 Ghz. Timing results are for processing 1 million closest points or 10 million ray intersections. Parallel scaling generally did not depend on the input mesh, as a single BVH is pre-computed and used for all queries. Scaling was poorer for very small query counts (~1000) due to thread overhead, but this did not effect the fundamental parallelism.

Our thread-based parallelism had no particular performance barriers, as queries are independent and do not mutate the BVH. We observed just about perfect linear scaling across all numbers of cores.

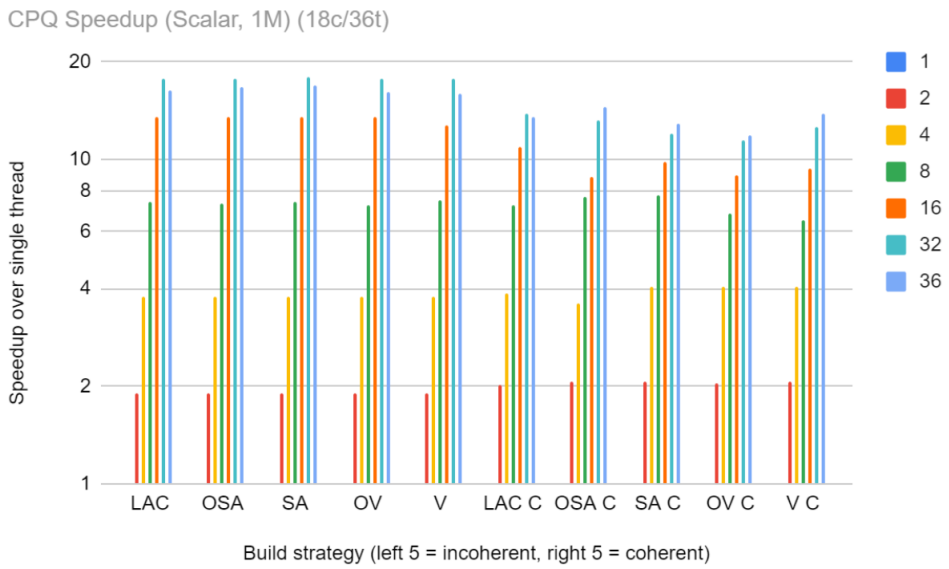

Figure 1: Thread scaling for CPQs

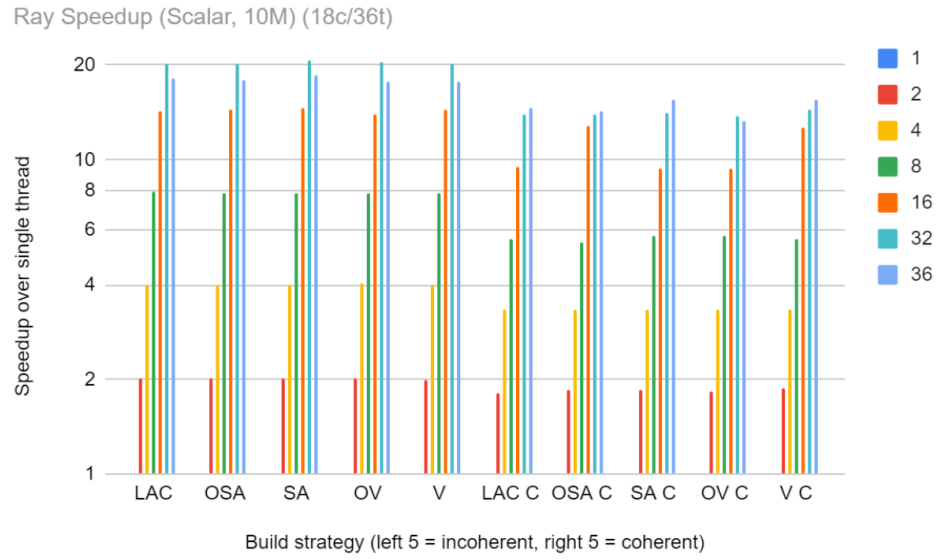

Figure 2: Thread scaling for Rays

Speedups were consistent across different BVH build heuristics, reaching about 12x at 16 threads and 16x on 36 threads (18 cores). Thread-based speedup was consistent for the scalar BVH case as well as 4-wide SIMD BVHs. This made sense given the test hardware, as the compute-heavy traversal process does not leave much room for the utilization of hyper-threads. Achieving good workload balance was not an issue.

More interestingly, we noted less speedup for coherent queries across the board when compared to incoherent queries. This was because incoherent queries naturally exhibited less coherent memory

access. When memory accesses are coherent between contiguous queries, arithmetic intensity goes up and the ability to use hyperthreads is further reduced.

The following graphs compare the performance of scalar BVHs and 4-wide SIMD BVHs/primitive tests:

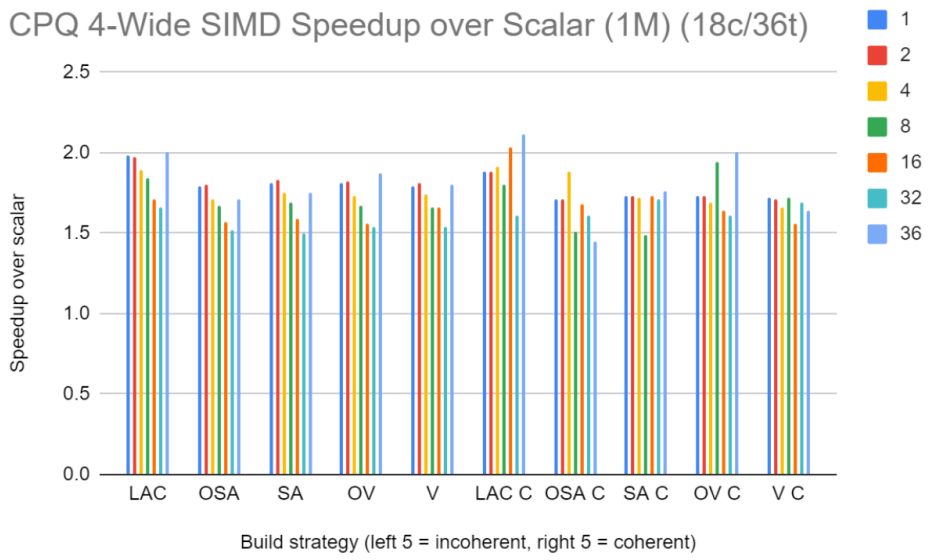

Figure 3: SIMD BVH scaling for CPQs

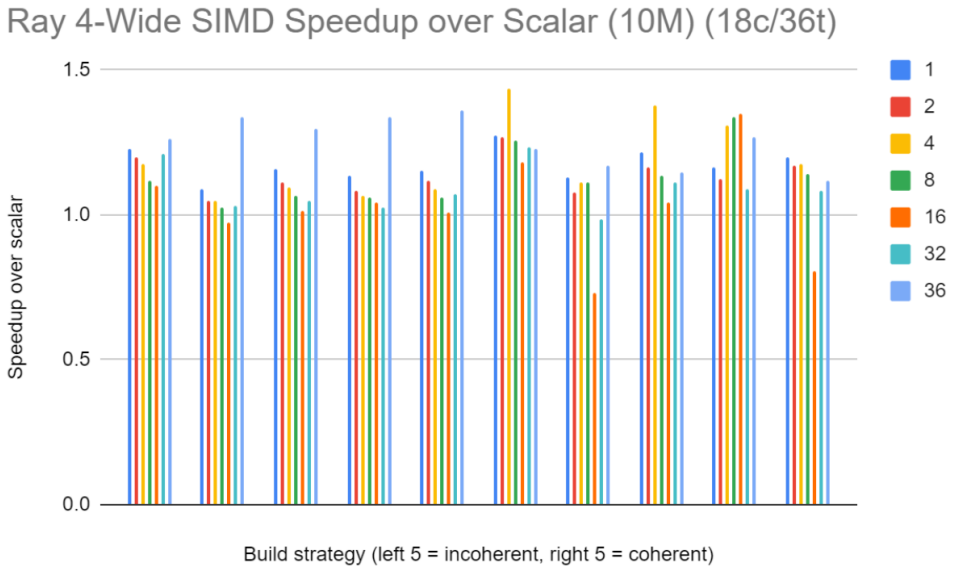

Figure 4: SIMD BVH scaling for Rays

This experimental setup was the same as the thread-based scaling tests. While these results (roughly 1.8x speedup for closest points and 1.25x for rays) may not seem particularly impres-

sive, they are consistent with previous results in high performance ray tracing, and exceed previous results for closest points. There are a few barriers to further scaling:

1. Vectorizing the BVH serves only to accelerate node traversal, and can never be fully coherent, since we are checking one point against four different bounding volumes. Unless all of the volumes are the same, they will require slightly different execution paths, so some lanes will be disabled for some tests.

   While vectorizing nodes allows us to check our query against all four children of each node at once, widening the tree actually creates unnecessary work: in the binary BVH case, we would check only a pair of nodes, and if one fails, its subtree would be cut before checking the next layer. In the wider case we have to check all options in the 'following' binary layer(s).

2. Vectorizing primitive checks within leaf nodes is actually more impactful than traversal and reveals the reason vectorization shows more benefit for closest points than rays. This is because closest points exhibit poor culling during traversal, leading to more time spent in primitive checks. We observed the time spent on primitive checks for CPQs as up to 60% of total execution time, whereas it was unusual for rays to exceed 20%.

   While vectorizing primitives also suffers from slight execution divergence when checking a single point against four different triangles, lane utilization is still high. This is because the primitives, being grouped by the BVH build, are spatially contiguous.

Comparisons against 8-wide SIMD BVHs are not reproduced here, as 8-wide generally did not produce significantly better results than 4-wide.

3.1.2 Bounding Volumes

We implemented binary BVHs using three additional types of node bounding volumes: oriented bounding boxes, bounding spheres, and sphere-swept rectangles. Benchmarks across these volumes are shown here:

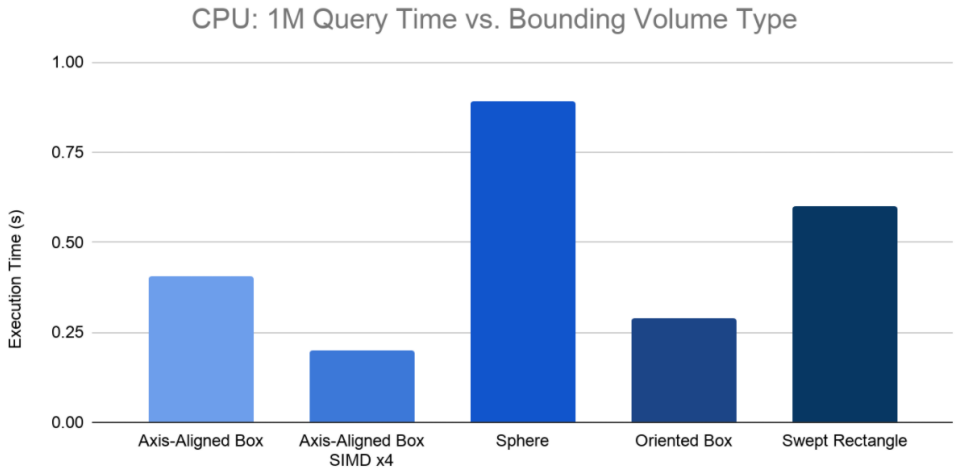

Figure 5: SIMD BVH scaling for Rays

Results for rays are not reproduced here, as for rays, axis-aligned boxes were the fastest across the board. A notable result for closest points, however, is the usefulness of oriented bounding boxes (OBBs). OBB nodes more tightly fit their associated geometry, leading to better culling behavior during traversal. Use of OBBs reduced the proportion of time spent in primitive queries for closest points to roughly 40%, and despite increasing the cost of checking whether a node must be traversed, lowered the overall runtime compared to axis aligned boxes. However, this behavior was dependent on the input mesh: while oriented boxes were always an improvement, the benefit was most apparent for geometry not suitable for axis aligned boxes (e.g. containing non axis-aligned extents like hair or thin structures).

3.1.3 Embree Comparison

We compared our scaling results against a production BVH library, Intel's Embree. The experiment setup was again the same as above.

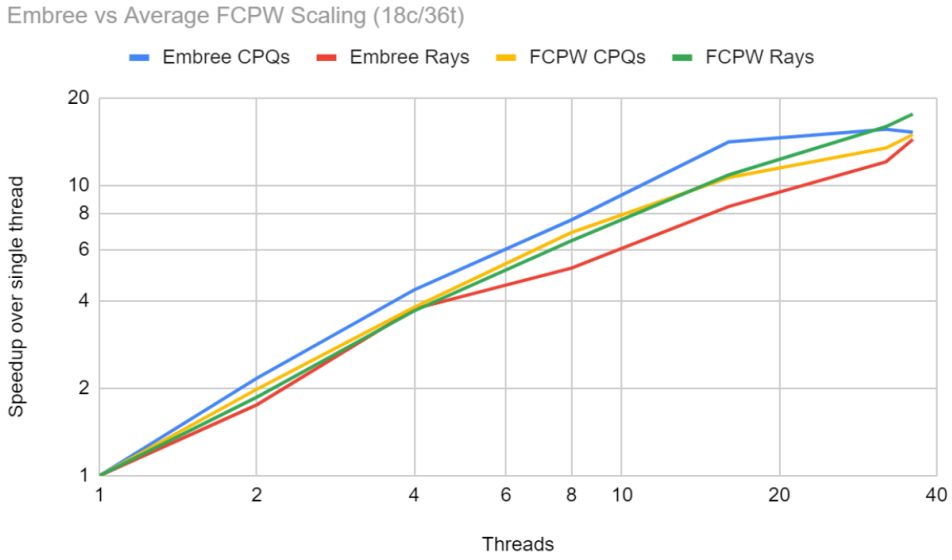

Figure 6: FCPW CPU vs Embree Scaling

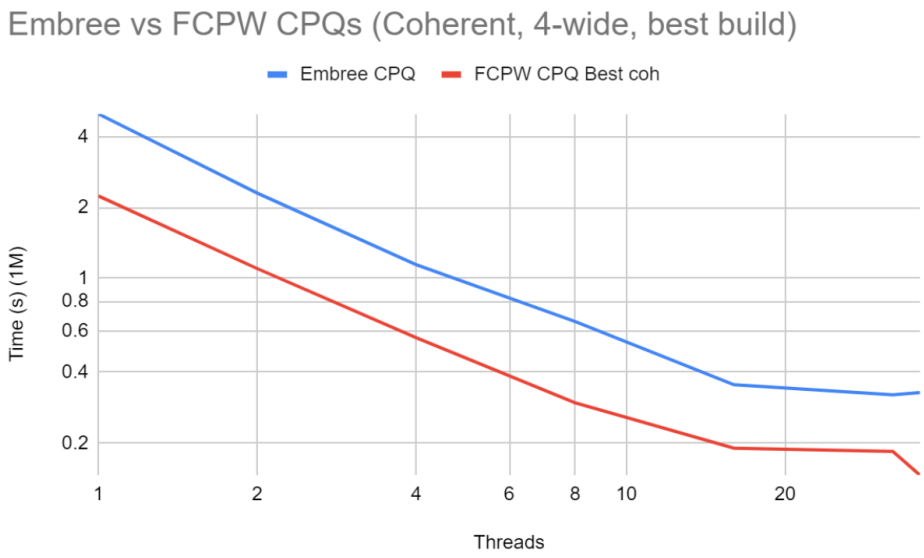

Figure 7: FCPW CPU vs Embree Scaling

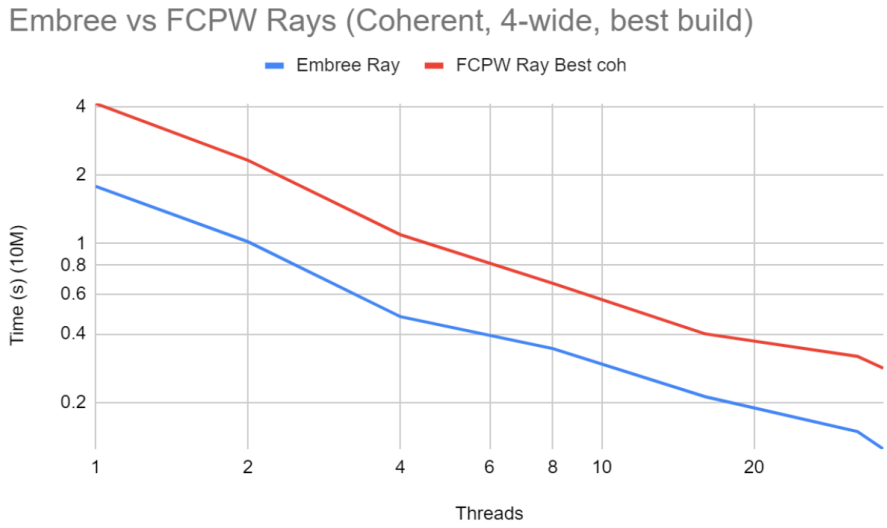

Figure 8: FCPW CPU vs Embree Scaling

As seen here, our thread scaling results approximate Embree's. More interestingly, our closest point queries are ~2x faster than Embree's across the board. This is due to our vectorization of closest point primitive computations, something Embree does not do. Our performance for rays lags Embree by a similar ~2x, but this may be attributed to the use of less advanced BVH construction algorithms.

3.2 GPU

On the GPU, the SIMD situation is pre-determined: a set of queries must be automatically partitioned into blocks that may be run on independent GPU cores. Each block is partitioned into 32-wide SIMD warps that run within a single GPU core.

Hence, we focused on comparing the performance of various BVH traversal and build strategies. The following graphs show performance results for both closest points and rays across three test cases. The first is a single cube: a small mesh where the BVH is insignificant. The second is the Stanford bunny: a medium sized mesh where the BVH is very significant, but is generally well-behaved for rays. Third, Sponza: a realistic atrium scene with far more triangles and spread-out geometry.

Timing results are again reproduced for 1 million closest point queries or 10M ray intersections, generated in the same method as the CPU tests. GPU benchmarks were run on an NVIDIA RTX 3090 running at ~ 1.7Ghz. Results were gathered via Vulkan performance counters that measured total time spent in each traversal compute shader. Execution time does not include memory transfers between the CPU and GPU. Performance information was validated via NVIDIA's Nsight graphics debugger compute shader profiling.

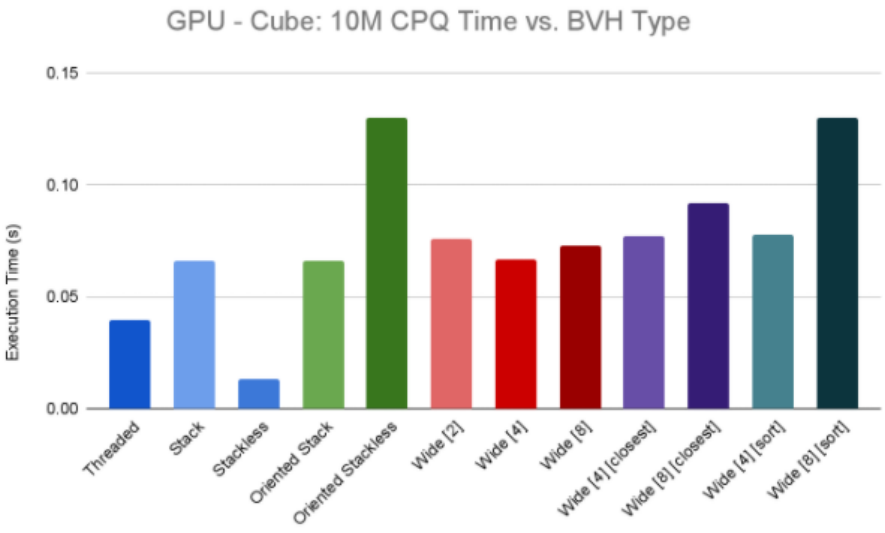

Figure 9: GPU Performance: Cube CPQs

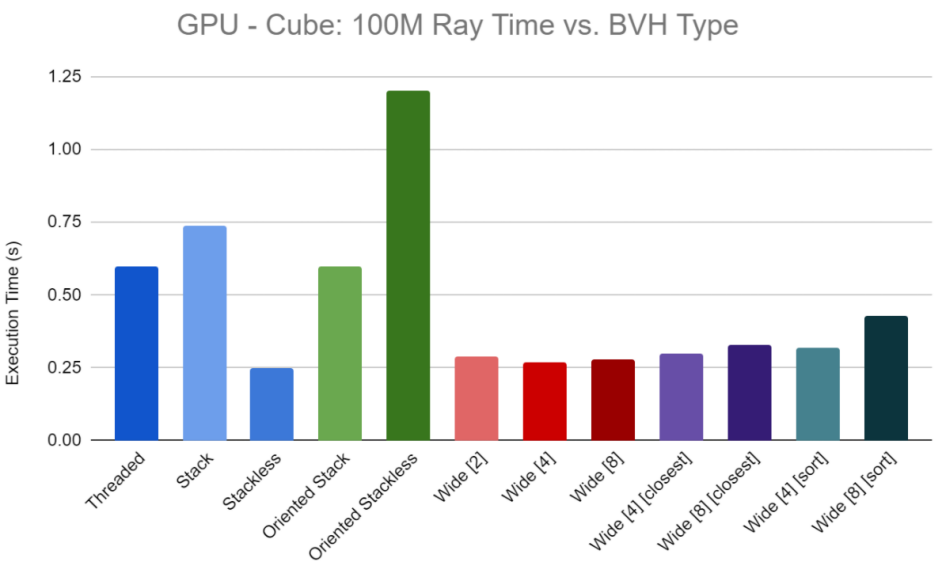

Figure 10: GPU Performance: Cube Rays

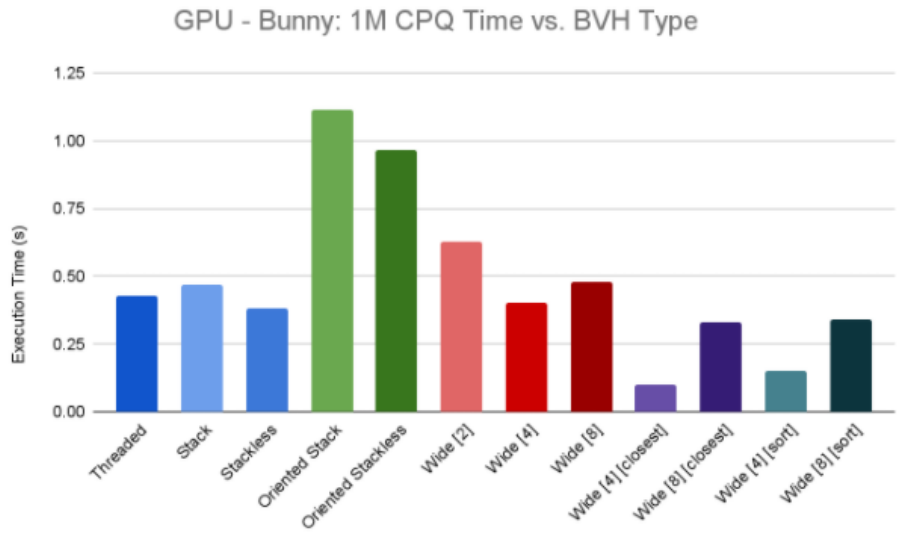

Figure 11: GPU Performance: Bunny CPQs

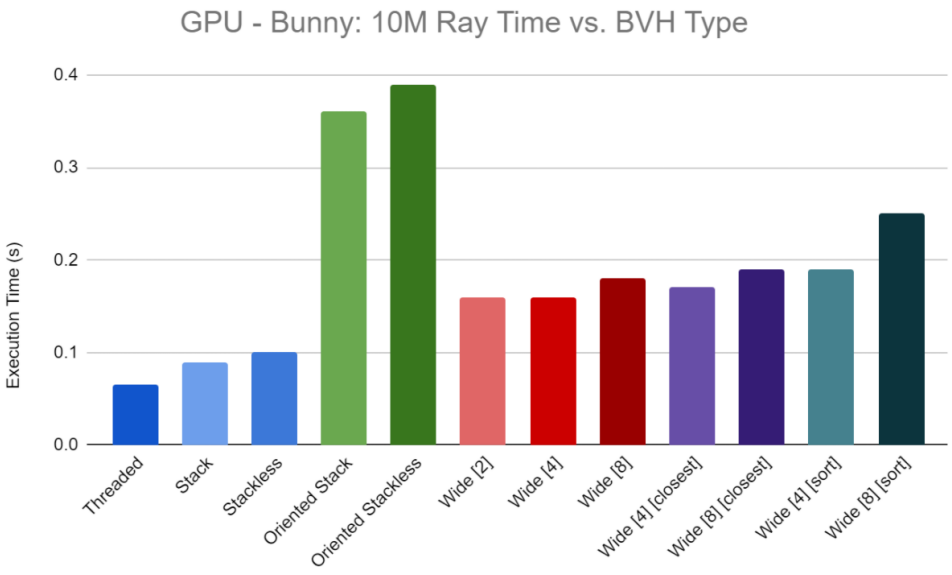

Figure 12: GPU Performance: Bunny Rays

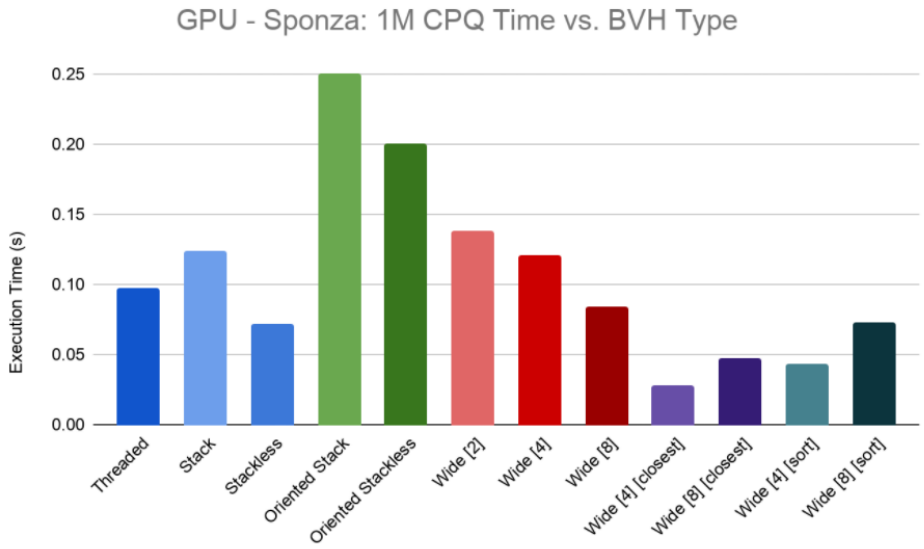

Figure 13: GPU Performance: Sponza CPQs

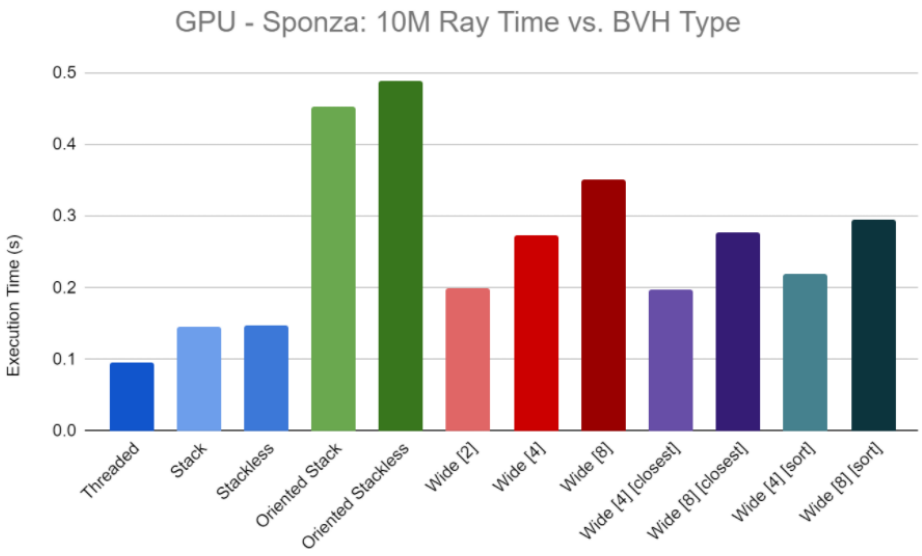

Figure 14: GPU Performance: Sponza Rays

For CPQs, although results varied between different strategies and tests, two common themes arose—execution divergence and poor culling behavior. These problems were anticipated from the CPU results, and were the major determining factors as to which methods performed better on which tests.

Even the best implementations failed to achieve high warp SIMD utilization, as even spatially coherent input queries tended to take different paths through the BVH during traversal. During primitive checks, all traversals exhibited lane occupancy between 30% and 40% (as reported by Nsight).

Similarly, poor culling meant that each query had to check a large subset of the primitives. This exacerbated the issues with divergence: since the set of primitives for each query was only partially overlapping, even more time is 'wasted' checking primitives contained in the union of all sets necessary for the warp. All traversals spent at least 50% of execution time processing primitive computations.

Beyond the behavior consistent across all GPU options, we noticed interesting behavior across each test dimension. We compared BVH type, test case, and closest points vs rays:

3.2.1 Test Cases

1. Cube

   Benchmarks were run on the cube in order to test scaling behavior when the BVH is irrelevant. There are so few primitives in the scene that the most efficient strategy is to simply test each query against every primitive. This gave the expected results: the lowest overhead traversal (explicit stack-less) gave the fastest results. All others introduced extra overhead. The same behavior was seen in the ray intersection case.

2. Bunny

The bunny served as a run-of-the-mill single-mesh test and did not exhibit any surprising behavior. Comparisons between different traversal strategies are given below. One aspect to note is that while doing ray intersections with the bunny is generally very fast, closest points become particularly slow to compute near the medial axis of the shape. This is because they are close to many disparate parts of the bunny, and hence queries may have to traverse nearly the entire BVH to find the truly closest primitive.

3. Sponza

   This served as a full-scene, distributed geometry test case. Its behavior was significantly different than the bunny across most traversals for CPQs, though the ray case was similar between the two test cases. Closest points were comparatively faster on this scene because there were comparatively fewer difficult medial axis cases: almost all regions were associated with local regions of the mesh.

3.2.2 Traversal Strategies

1. Stack vs Stack-less

   The stack-less traversals generally performed better than the stack based ones, at least when discounting the additional benefit of wide tree distance optimizations. The stack-less approaches increased warp occupancy by about 7% by enforcing a single traversal order through the tree. Further, these traversals reduced register pressure by eschewing dedicated memory for an explicit stack. Nonetheless, these effects were not especially large, as again, most time was spent in per-primitive closest point tests.

2. Oriented Boxes vs Axis Aligned Boxes

   The benefit (or detriment) of using oriented boxes depended strongly on the test case. For all three tests, OBBs hurt performance due to increasing the cost per node traversal without improving culling enough to compensate. This was because most of the geometry in these test cases could be fit by axis-aligned boxes. However, for the Sponza scene, oriented boxes were not *too* much of a detriment, suggesting that oriented boxes may perform better with particularly degenerate geometry. Another benchmark was run on a poorly distributed subset of the Sponza scene (the foliage meshes), and we found that the oriented box case was actually faster than axis aligned boxes, with the OBB stack-less traversal matching the speed of 4-wide closest-first traversal.

   Also note that OBBs are never useful for rays, as performance in those cases depends far more on traversal speed than efficient primitive culling. This is because rays naturally cull most BVH nodes by simply not intersecting their bounding volumes. A closest point, on the other hand, never has the ability to "miss."

3. Tree Widths

   Our wide BVHs typically provided mild benefits when compared to the basic binary BVH approaches. As discussed in the CPU section, widening the tree creates some amount of unnecessary work. However, on the GPU, keeping the tree shallower makes a traversal's chain of dependent memory accesses shorter. This increased occupancy by about 14%. However, most (90%+) memory accesses hit L2 cache due to the spatially contiguous BVH tree layout, so the memory benefits were not large. Gains depended on the test case and tended to fall off after the 4-wide case. 16-wide trees were also benchmarked and found to be unilaterally worse than 4 and 8 wide trees.

4. Wide Ordering Optimizations

   The traversal ordering optimizations available for wide trees (traversing to the closest child first, or sorting all children in order of distance) helped CPQs significantly due to improved culling. Despite the extra node bounding volume checks due to widening the tree, being able to pick out the closest node from the next two 'binary' levels of the tree let us much more effectively cull more distant children. However, doing a full sort of the children by distance was typically not worthwhile, especially as tree width increased. Sorting simply added too much computational overhead during traversal.

3.2.3 Rays vs CPQs

The story for rays was very straightforward: stack-less traversal of a binary BVH built from axis aligned boxes is the best in all cases. This agrees with prior work on high performance GPU ray-tracing.

The story for CPQs is far more varied, and depends strongly on the nature of the input mesh. This is because CPQs require traversing much, much more of the BVH tree, and hence end up being 10-100x slower than ray intersections.

3.2.4 Coherent Input

Lastly, we also ran benchmarks with explicitly randomized input queries. This decreased performance across the board by 30%-300% due to decreasing coherence. As expected, less coherent inputs lead to more divergent execution. However, this did not make *as much* of a difference as previous work has observed for ray intersections. This was because for CPQs, there is limited coherence to disrupt in the first place. Incoherent CPQs ran on the bunny gave the following result (other results omitted as they follow the same pattern):

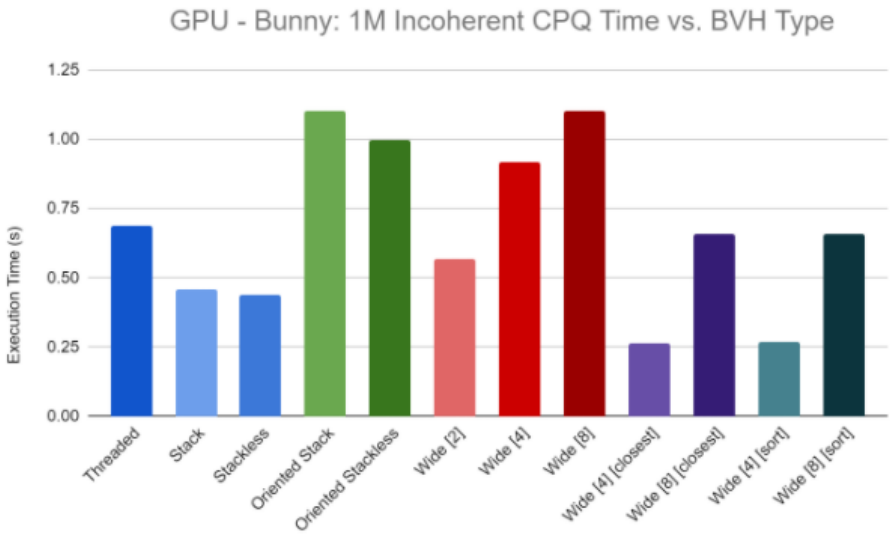

Figure 15: GPU Performance: Incoherent Bunny CPQs

3.3 GPU vs CPU

Comparing CPQ behavior across our CPU and GPU implementations is not an entirely apples-to-apples comparison, as the parallel strategies employed on the two types of processor are quite different, and hence respond differently to different inputs. In the best case (smart traversal algorithm, suitable input), the GPU implementation exceeds the performance of our CPU implementation running on 32 threads by 2-10x. In the worst case (poor traversal algorithm, divergent inputs), the GPU implementation lags behind the CPU by a similar 2-10x. Of course, if we limited the CPU implementation to one thread, the GPU would be up to 300x faster.

4 Lessons Learned

Academically, it was interesting to explore various BVH construction and traversal strategies via implementing them, as opposed to simply reading papers. It was similarly enlightening to reason about the various performance tradeoffs on the GPU, though we did not find a panacea. I also learned how to work with Vulkan and its new hardware ray tracing APIs, which will be useful for future work. I don't believe there were any major surprises, as we had some idea of what would (and wouldn't) be useful from Rohan's previous work on the CPU side. However, the disparity between various GPU strategies was bigger than expected and emphasized the importance of choosing the right one for a given problem.

In general, I also learned that research is difficult to do as a busy undergrad (shocking, I know). I wouldn't recommend taking 3 technical, project based classes and working as a TA on top of a research project. But I do need to graduate... Similarly, staying connected when working alone from home can be challenging, and I probably should have asked for more guidance more often. Nonetheless, I think we have produced some interesting results, and I hope to do research work next semester as well.

5 Conclusions and Future Work

On the CPU side, our work confirms that most performance tuning for high performance ray tracing also applies to CPQs. For example, we saw significant gains using vectorized trees. However, unlike in the ray case, we found that using more complex bounding volumes such as oriented bounding boxes often improves performance due to increased culling.

On the GPU, we find that no BVH-based approach gives orders of magnitude improvement over the CPU implementation, as all struggle with execution divergence. Such divergence is particularly harmful to performance due to poor culling, as every query in a vector lane must check many disparate primitives. However, we note that some GPU approaches are very viable, focusing on the use of both stack-less BVH traversals and wide traversals with ordering optimizations.

In the future, we would like to develop our findings into a broadly useful open-source library for computing closest point queries. This would be useful for many other projections and applications, as mentioned in the problem overview. Further, it would be interesting and impactful to explore the use of hardware ray-tracing cores for the closest point problem. Contemporary work has been published on the use of ray queries for other problems such as tet-mesh point location, force-directed graph simulation, and k-nearest-neighbor finding, so we believe this is a promising avenue.

6 References

References

[1] Attila T. Áfra, Ingo Wald, Carsten Benthin, and Sven Woop. Embree ray tracing kernels: Overview and new features. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 145–149, New York, NY, USA, 2009. Association for Computing Machinery.

[3] Benedikt Bitterli and Simon Kallweit. Optimizing query time in a bounding volume hierarchy. 2016.

[4] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, page 1225–1233, Goslar, DEU, 2008. Eurographics Association.

[5] Manfred Ernst and Gunther Greiner. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 35–40, 2008.

[6] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, page 7–12, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 7–13, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 33(4), July 2014.

[9] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In Vlastimil Havran and Karthik Vaiyanathan, editors, *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2017.

Open-Source Libraries:

1. Enoki (CPU vectorization)
2. Embree (for CPU comparisons)
3. ImGui (graphical user interface)