15618 Spring 26

Parallel BVH Construction and Traversal for Accelerated Ray Tracing

Final Project Report

Names: Soham Narendra Joshi, Viren Dodia

Andrew IDs: sohamnaj, vdodia

Project Website: https://sohamjoshi.github.io/parallel-bvh-raytracer-project/

Github Repo: https://github.com/SohamJoshi/parallel-bvh-raytracer-project

TABLE OF CONTENTS

SUMMARY 2
BACKGROUND..3
RAY TRACING WORKLOAD 3
BOUNDING VOLUME HIERARCHY (BVH) & DATA STRUCTURES 3
KEY OPERATIONS 5
Why BVH Construction is Hard to Parallelize 5
LOCALITY...5
PIPELINE OVERVIEW..5
Dependencies and Parallelism 6
Inputs and Outputs 6
APPROACH..6
FINAL DESIGN OVERVIEW..7
Technologies and Target Machines 7
Mapping to Parallel Hardware 7
Changes to the Serial Algorithm..8
CPU BVH Construction Details 8
Flat BVH and GPU Traversal...8
Optimization Iterations 9
What Did Not Work / Less Effective Attempts 11
RESULTS...11
Experimental Setup..12
CPU BVH Build Scaling 12
CPU Optimization Progression 13
CPU Microarchitecture Analysis (perf stat) 14
Final Hybrid Performance 15
Hybrid Performance Breakdown 16
Effect of Problem Size / Scene Complexity 17
OTHER RESULTS...18

Frontier Hybrid Tradeoff ... 18
GPU Profiling ... 19
Task Threshold ... 20
What limited speedup? ... 20
Was the machine target sound? ... 21
OUTPUT IMAGES ... 21
WORK DISTRIBUTION ... 23
REFERENCES ... 23

SUMMARY

We implemented and tested multiple parallel techniques for building and traversing BVHs for large ray tracing applications. Our implementation supports triangle meshes (such as the Stanford Bunny and Dragon models), builds BVHs using OpenMP on the CPU, and traverses the BVH on the GPU via CUDA. Improvements made to the top-down BVH construction method include task parallelism, caching of primitive information, bucket partitioning, and atomic node allocation in a flat array structure. The final implementation achieved an end-to-end speedup of around 27 times compared to the optimized CPU version on the Stanford Dragon (871K triangles) at 4K resolution.

BACKGROUND

RAY TRACING WORKLOAD

Ray tracing renders an image by casting rays from the virtual camera through every pixel and determining where that ray first intersects a surface in the scene. Given triangle meshes, the naive algorithm checks all triangles against every ray cast which is extremely inefficient because its computational complexity is O(R x N), where R represents the number of rays and N is the number of triangles. In other words, for example a 4K render of the Stanford Dragon mesh with 871,000 triangles, would end up taking about 8x10^12 checks per frame.

The program requires three basic parameters: a triangle mesh (given in PLY format), camera information, and image resolution. The program will output a color value for every pixel of the image stored in PPM format. Ray-triangle intersection is the most computationally expensive step and can be sped up significantly using spatial indexing techniques.

BOUNDING VOLUME HIERARCHY (BVH) & DATA STRUCTURES

This structure represents a tree containing all the scene primitives as shown in figure 1. The AABB of an internal node is guaranteed to contain anything in its subtree. Each leaf stores a small number of objects/triangles. For ray tracing, we test whether a ray intersects the bounding box of some node. We can immediately skip the entire subtree if there is no intersection. Otherwise, we process the two child nodes recursively. Thus, the average cost of ray intersections becomes O(log N) instead of O(N) as shown in Figure 2.

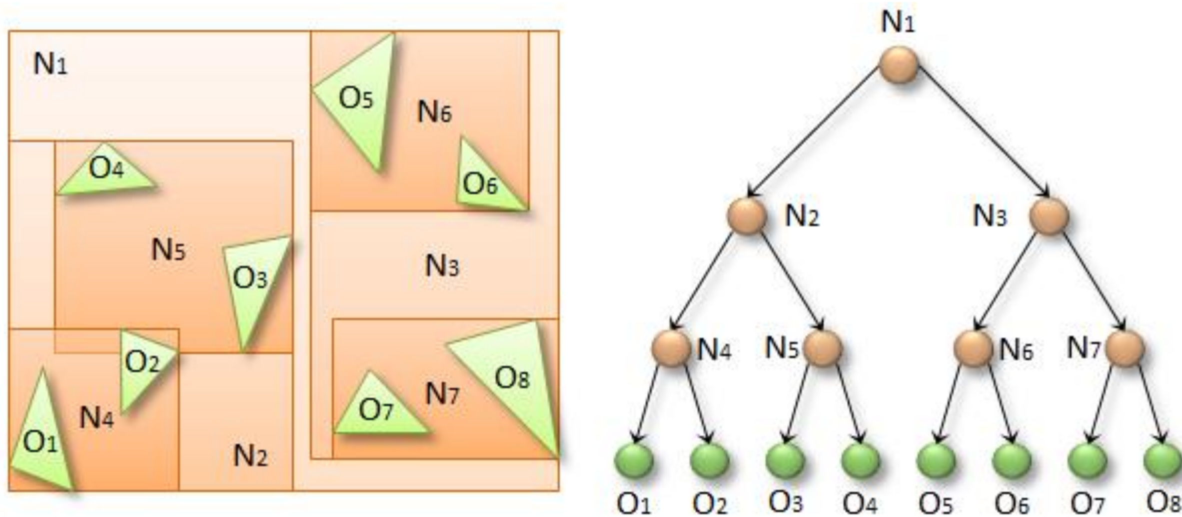


Figure 1: A bounding volume hierarchy [1]

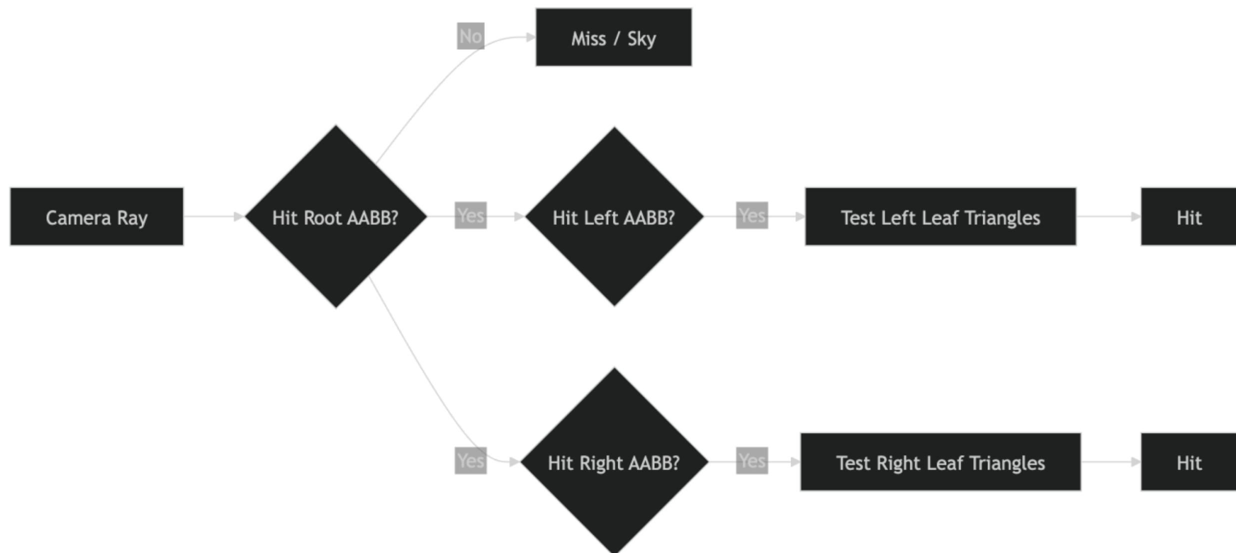


Figure 2: BVH Traversal

All of the data structures includeed in our project:

1) An array of primitives in our case are triangles. Their bounding boxes and centroids should be pre-calculated.

2) A tree of BVH nodes with a bounding box and a pointer or a range of primitives (for a flat version) at each node.

3) In GPU implementation, a flat array of nodes with primitive ranges and integer references to children instead of pointers.

KEY OPERATIONS

- **Primitive metadata computation:** computing triangle AABBs and centroids.
- **BVH construction:** recursively compute node bounds then choosng split axis then partition primitives and finally creating child nodes.
- **BVH traversal:** test ray against node AABBs, check if we hit children, and test triangles in leaves.
- **Ray-triangle intersection:** compute closest triangle hit for each ray.
- **GPU traversal**: process one ray/pixel per CUDA thread over the flat BVH array.

Why BVH Construction is Hard to Parallelize

BVH construction is inherently recursive and non-uniform, making efficient parallelization difficult. For instance, a parent node must first calculate its bounding volume and select an appropriate split axis before the two sub-trees under the parent node can be built independently. In addition, the sub-tree sizes post-split are rarely balanced, making dynamic load balancing difficult with a fixed number of threads in a thread pool.

Unlike the construction process, however, ray tracing is data parallel. Each pixel query is independent of other queries, requiring no synchronization in the traversal process, which is a read-only process on the BVH data structure.

It is this difference between the processes, where construction is control-intensive and non-uniform while ray tracing is data parallel and uniform, that motivates our hybrid CPU-GPU solution. For the data-parallel problem of ray tracing, the level of parallelism is determined by the number of rays required to trace. For an image of size WxH pixels, we have WxH independent rays.

LOCALITY

There are important locality issues involved with the load. Due to poor locality in cases where pointers are used to create BVHs, we used a simple BVH structure with integer-based child indices that is stored in contiguous arrays. The internal nodes at the top of the BVH tree are reused by several rays does making them good for GPU SIMT, whereas the pattern for leaf nodes is irregular. However, there is high divergence between rays that follow different paths through the BVH tree so utilization will be low.

PIPELINE OVERVIEW

Figure 3 shows our final hybrid pipeline consisting of full BVH construction and traversal. The primitives go through a sequence of stages where the initial stage is the mesh loading stage, then the stages of the BVH tree building algorithm to the final ray intersection stage.

Figure 3: Final Hybrid Pipeline

Dependencies and Parallelism

One of the biggest dependencies in BVH building is that parent has to calculate the bounding box of primitivs and then make a split before building the nodes in both subtrees. Having performed this split, both subtrees become completely independent from each other and can be constructed simultaneously. However, traversal has much more data parallelism due to the possibility to handle all rays or pixels independently. At the same time, inside one single ray, the process of traversal becomes completely sequential due to dependencies between nodes.

Inputs and Outputs

Input to the system includes scene, acceleration structure type, and image resolution. In our experiments, the primary scenes are triangle meshes stored in PLY file formats, such as the Stanford Bunny[3], Dragon[4], and Drill[4]. The user specifies the scene and acceleration method via command-line arguments, e.g., dragon bvh_omp_opt or dragon bvh_cpu_gpu_flat. Additional runtime arguments include the width and height of the image, shading model, OpenMP task threshold, and validation.

Output from the system is a rendered image in PPM format, where the color of each pixel is an RGB value from the closest intersection between a ray and a surface. During performance evaluation, the program also provides timing information on operations such as BVH construction, host-to-device memory copy, CUDA kernel traversal, and device-to-host memory copy.

APPROACH

FINAL DESIGN OVERVIEW

Our final approach is a hybrid CPU–GPU BVH pipeline where the CPU construts an optimized flat BVH using OpenMP task parallelism as well as the GPU traverses the flat BVH using one CUDA thread per pixel. This split is done so it matches the workload structure because BVH construction is recursive and irregular, while traversal is massively data-parallel across rays. Our final hybrid pipeline is:

Figure 4: Final Pipeline

The implementation has command-line acceleration options for CPU and hybrid modes like bvh_omp_opt, bvh_cpu_gpu, and bvh_cpu_gpu_flat. The application also takes input arguments such as image dimensions, shading type, OpenMP task threshold, and correctness verification during runtime.

Technologies and Target Machines

The algorithm was implemented using C++ and CUDA programming languages. For constructing the BVH, we utilize OpenMP in order to create the data structure in parallel on CPU and use CUDA in order to do the tree traversal on GPU. All experiments are conducted on GHC computers (e.g., GHC28, GHC32).

Mapping to Parallel Hardware

The primitive array works with the CPU threads in preprocessing. Here we process a subset of triangles by OpenMP thread to calculate AABBs and centroids.

In BVH construction, recursive subtrees are directly being mapped to OpenMP tasks. Once the split of the node is done, the left and right children ranges are then constructed independently. For BVH traversing on the GPU, the node array and primitive array are copied to the CUDA. Here, each CUDA thread deals with only one ray per pixel. The traversal of read-only BVH nodes happens independently for each CUDA thread and tests triangles at leaf nodes. Here, BVH nodes follow a sequential storage scheme in the array. The below figure shows our mapping:

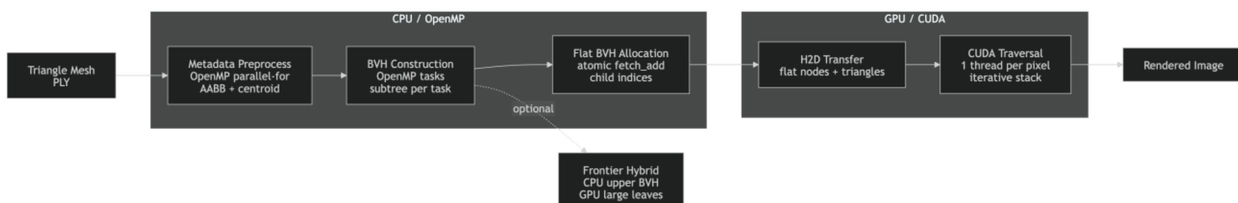


Figure 5: Hardware Mappings for each component in pipeline

In the flat BVH build process multiple OpenMP tasks can create nodes concurrently as we use an atomic fetch_add counter for indices in the shared flat node array.

Changes to the Serial Algorithm

Yes we made several changes to the serial algorithm in order to make the algorithm more parallel. Initially, we switched from recursion for vector copies to using ranges for constructing an array of primitives. Secondly, we added caching for primitive metadata to avoid dominating construction time due to multiple bounding box and centroid computations. We then reduced the cost of splits by removing full sorting in favor of nth_element and utilizing bucket-based partitioning. Lastly, we converted our pointer-based BVH to a node-array form that works well on GPUs.

In summary, all the above modifications were made without altering the overall high-level BVH top-down algorithm, only improving its parallelism-friendly nature. The range-based BVH construction approach improved CPU efficiency, task-based recursive partitioning identified subtree parallelism, bucket-based splitting decreased split costs, and the flat node structure facilitated GPU traversal using CUDA.

CPU BVH Construction Details

Our CPU BVH implementation uses a top-down recursive construction strategy. Each node computes the bounding box of its primitive range then selects the longest axis, partitons primitives using a bucket-based split, and recursively constructs children. OpenMP tasks are then spawned for sufficiently large ranges, controlled by a task threshold. For small ranges, construction proceeds sequentially to avoid excessive task overhead.

Pseudocode:

```
build(start, end):
      compute bounds for primitives[start:end]
      if range size <= leaf size:
            create leaf node
      return
      axis = longest axis of bounds
      mid = bucket_partition(start, end, axis)
      spawn task for one child if range is large
      build other child locally
      wait for child task
```

Flat BVH and GPU Traversal

Since a GPU cannot iterate over a BVH constructed using CPU pointers, nodes are stored in a simple array format. Each node contains the axis-aligned bounding box, indices of its two children, a flag to denote whether it is a leaf node or not, and the range of primitives it contains. Both of these structures are copied to GPU memory along with the triangles.

In the bvh_cpu_gpu implementation, the CPU first builds an optimized BVH and then flattens it. In the bvh_cpu_gpu_flat implementation, the CPU builds directly into a flat GPU-ready layout, avoiding a separate flattening phase thus saving time.

Optimization Iterations

We arrived at the final design through several iterations:

1. Baseline Implementation
   Our first implementation was based on a sequential ray tracer as well as a top-down BVH. The initial BVH used a purely recursive algorithm in which we calculated the bounding box for all primitives within the current node's index range, picked the largest axis, sorted the primitives based on their position along the chosen axis, found the midpoint, and called the recursive function for both sides. We then built on top of this algorithm to establish a correctness baseline and a performance lower bound. Our implementation of the core ray tracing math such as ray trace, sphere and triangle intersections, shading, and image rendering is inspired by Peter Shirley's "Ray Tracing in One Weekend"[5] series but includes our own project modifications.

2. CPU Parallel BVH Construction: Six Iterations
   - Naive OpenMP Task Recursion
     In our first parallel approach, we used OpenMP task directives to create left and right subtrees in parallel after each split. This revealed parallelism at the subtree level across multiple CPU cores. But the upper levels of the tree are still built sequentially since the split is singlethreaded. Also, there are only two tasks at the top level. Speedup was not that great as the work at upper levels is small and the hot path is still dominated by serial split computation that is sorting.

   - Range Based Recursion
     The original code had an O(N log N) reduction in heap allocations per each recursive call, where the builer allocated a std::vector of primitive values for each node in proportion to the subtree size. The new implementation perform operations on ranges [start, end] on a single shared array of primitive types.

- Cached Primitive Metadata
  Our tests showed that the comparator used by std::sort would call bounding_box() and centroid() on each primitive whenever they were compared, which could happen millions of times during the tree construction process. To fix this bottleneck, we implemented a pre-processing step to calculate the AABB and centrid for each primitive only once, before the construction starts, and store these values in a cache array.

- nth_element Instead of Full Sort
  The initial implemntation sorted the primitive perfectly based on their centroids in each node. Sorting takes N log N time but we needed to find the median here which is a overall weaker constraint. Using nth_element() from the STL library instead of sort(), we partition the range such that the median is the pivot in O(N) time.

- Bucket Partition
  Next we the same computation of the median for a bucket-based split. In our tests using B=8 buckets on the Dragon benchmark, bucket splitting cut down the build time by about 28% to nth_element while achieving equally traversable BVHs.

- Flat BVH Layout for GPU Transfer
  Nodes for pointer-based BVH trees are not possible to be uploaded to GPU's memory since CPU virtual addresses have no meaning in device's memory. Therefore, we developed another approach for representing nodes which a flat array, where each node contains two integers that point to the location of their children in the array instead of pointers and so it can be sent to GPU's memory in order to perform ray tracing.

3. GPU Ray Traversal (CUDA)
   In our CUDA traversal kernel, we assign a single thread to each pixel. Our thread creates the ray from the camera, iteratively walks the flattened BVH using a local stack, checks only triangles at leaf nodes, and outputs the color of the closest hit. Because our BVH and our triangle array are immutable in the course of traversal, we do not need any synchronization between our CUDA threads. Our main concern on the GPU side is divergence, but our problem is very suited for GPU computing because of its many parallel rays.

4. Frontier Hybrid Variant (Experiment)
   We also performed an experiment on the design and evaluation of a
   hybrid approach. In this approach, the CPU constructs the higher layers of the BVH tree,

terminating when the number of nodes left is less than the size. Then, the GPU takes the incomplete BVH tree, and for each node, the GPU performs brute-force processing of all triangles contained in the frontier node rather than performing traversal. This was to find the balance between minimizing the build time by constructing the tree less, and maximizing the workload of each thread by testing more triangles in leaves. We have put the results in our analysis section.

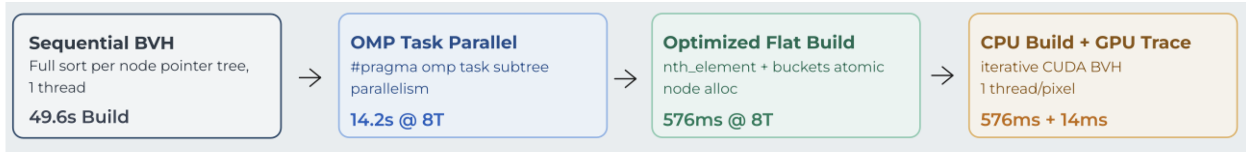


Figure 6: System design Evolution

What Did Not Work / Less Effective Attempts

Other methods for parallelization were not very effective. Threshold tuning did not have any effect after the division when split and metadata costs started to be become dominant part. Parallel bucket partitioning was not possible due to overhead. The hybrid frontier method helped decrease CPU costs of BVH creation however the increase in the number of leaves resulted in slower triangle testing on the GPU and we couldn't implement a optimized version for it so we left it.

RESULTS

Experimental Setup

Time is reported in milliseconds using scoped timers for each major pipeline stage. CPU tests have separate timers for BVH construction and rendering. Hybrid tests have separate timers for CPU BVH construction, host to device memory copy, CUDA kernel traversal, and device to host memory copy. Time to write output image to PPM files is omitted from timing statistics since image serialization is disk I/O and not part of our project goals.

Experiments were conducted on GHC machines. The primary test scene is the Stanford Dragon model imported from PLY and has 871,414 triangles. Tests are performed at resolutions of 800x450, 1920x1080, and 3840x2160. CPU tests used OpenMP with 1, 2, 4, and 8 threads.

Scene	Vertices	Triangles	Purpose
Bunny	35,947	69,451	medium mesh
Dragon	437,645	871,414	large mesh / main benchmark
Drill	881	1,288	small mesh for quick check

CPU BVH Build Scaling

The optimal CPU BVH construction algorithm shows a sub-linear relationship between the number of threads used and execution time. On the Dragon dataset, the construction time dropped from 2687 ms for one thread to 570 ms for eight threads, indicating a 4.71x speedup. This validats the assumption that recursive subtree building reveals some opportunities for parallelization however, the speedup is bounded by serial processing within each node. The Results and graphs (figure 7 and 8) are shown below:

Threads	Build Time ms	Speedup
1	2687	1.00x
8	570	4.71x

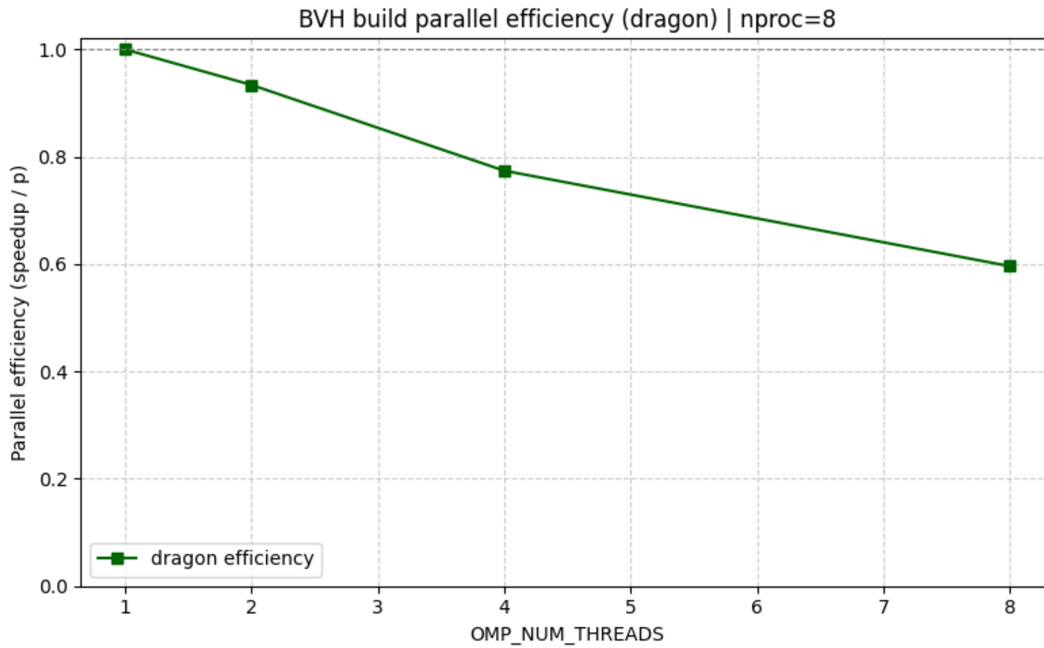


Figure 7: BVH Build Parallel Efficiency for Dragon Scene from 1 to 8 threads

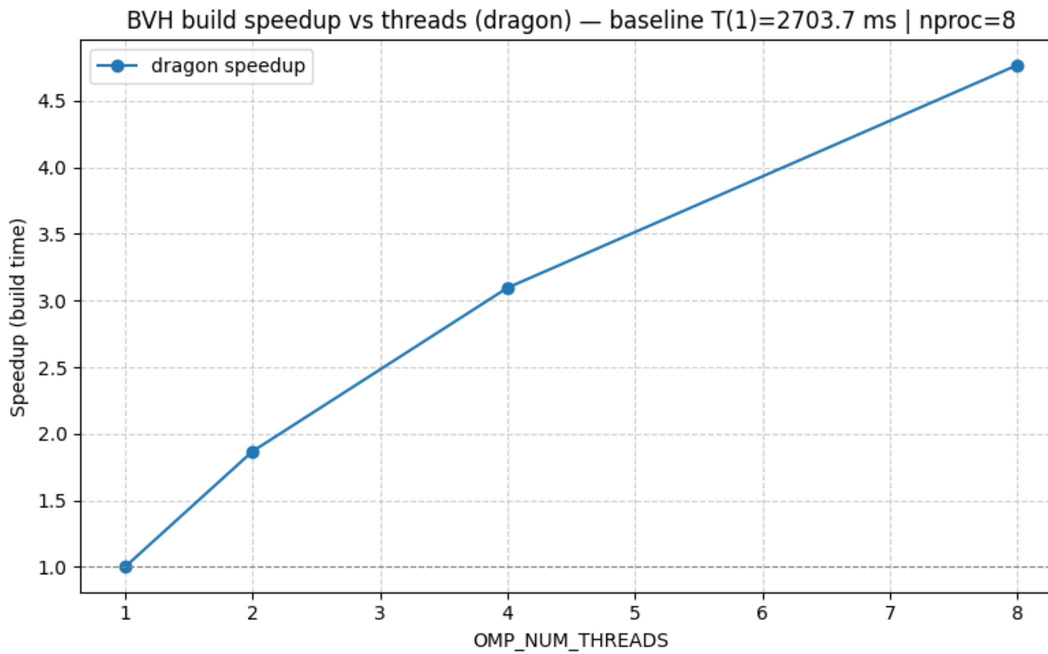


Figure 8: BVH Build Speedup vs Threads for Dragon Scene

CPU Optimization Progression

Improvement came not just from implementing OpenMP tasks but also from reducing the effort involved in doing so at each node. In the initial full-sort BVH builder implementation, much time was being spent in sorting operations as well as re-computation of metadata for primitives. Using range recursion eliminated the problem of recursively copying the vectors, use of caching prevented the recalculation of AABB or centroid calculations, and use of bucketing minimized split overheads.

As shown below in the figure 9 we focused on eliminating or replacing sequential parts with parallel friendly components and we saw really good results.

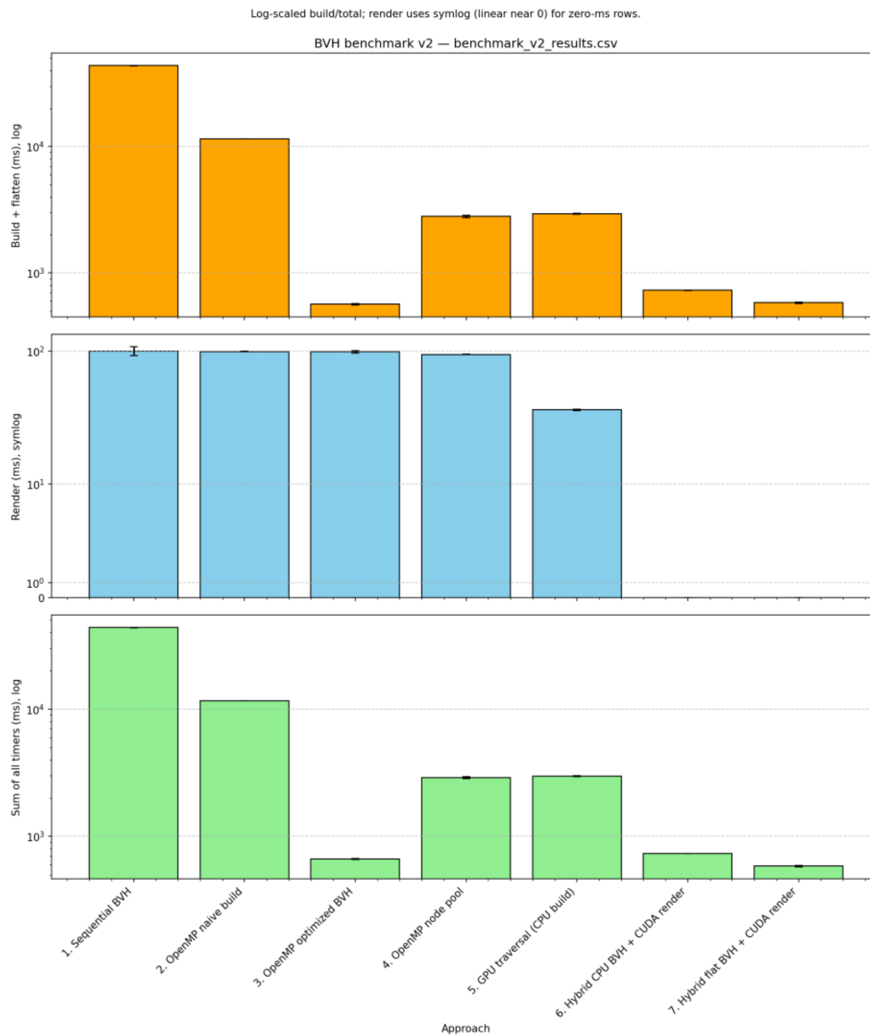
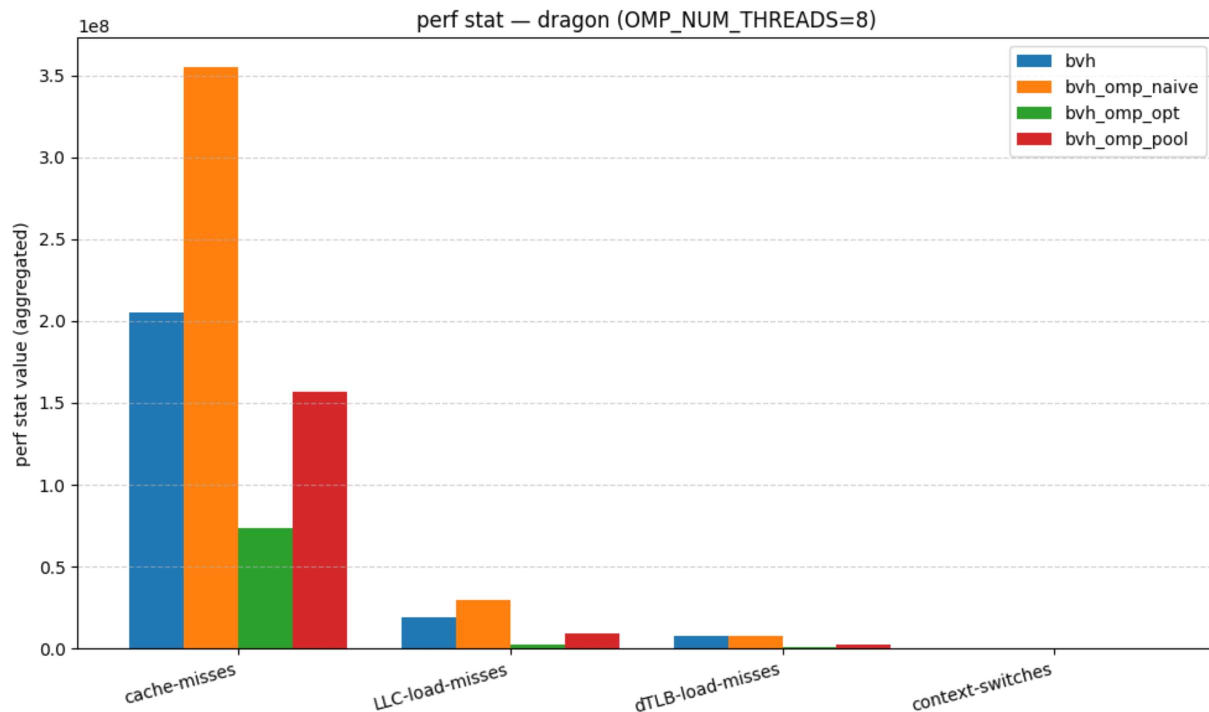


Figure 9: Comparsion of BVH Build optimizations

CPU Microarchitecture Analysis (perf stat)

Beyond wallclock, we used perf stat on GHC76 (i7-9700, 8 cores, 12 MiB L3) to measure where each CPU variant's cycles were going.



Some observations that stand out:

bvh_omp_opt executes 11.2x fewer instructions and incurs 2.8x fewer cache misses than the sequential baseline. The bucket partitioning, nth_element median split, and cached primitive metadata (described in the Approach section) reduce algorithmic work, not just parallelize it. This is why the wall-time win exceeds the 4.7x a pure 8-thread parallelization would deliver.

bvh_omp_pool regresses by 2.9x vs. bvh_omp_opt despite using the same recursive structure. perf shows it issues 3.3x more instructions and 2.1x more cache misses. perf c2c traces (collected with perf c2c record/report) confirm that the atomic fetch_add node-allocation counter is a hot HITM line, indicating cache-line bouncing between cores — the pool's serialization hot spot is the source of the regression.

bvh_omp_naive provides almost no benefit over sequential because it parallelizes only the leaf-level work; the dominant surrounding_box / sort cost still serializes on the recursive descent. This motivated the optimizations in bvh_omp_opt.

Final Hybrid Performance

The hybrid approach provides substantial gains in end-to-end compute time at higher resolutions. At 4K, the tuned CPU rendering engine required 16.7 s (BVH construction + CPU traversal), whereas the final hybrid pipeline required only 621 ms. This represents a 26.9x gain in end-to-end compute time. **The compute speedup is larger at higher resolutions** because the amount of traversal computation increases with the number of pixels, and traversal maps well to the GPU.

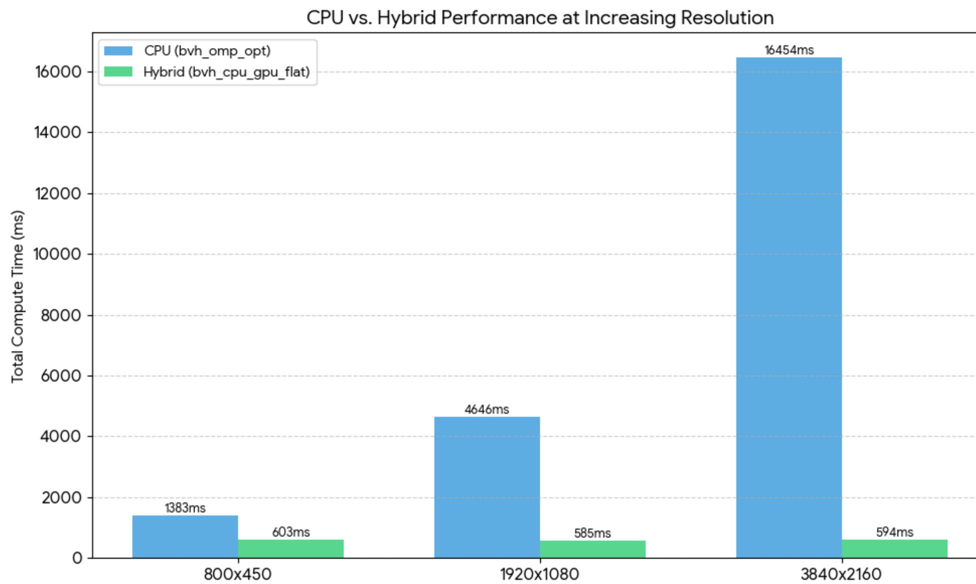


Figure 10: CPU vs Hybrid Total Compute Time by Resolution

Hybrid Performance Breakdown

In the hybrid 4K breakdown, there is an important bottleneck transition. The traversal using CUDA is just 14 ms, whereas CPU BVH construction in a flat manner is around 579ms.

Stage	Time ms
CPU Flat BVH Build	579
H2D Transfer	5
CUDA Kernel	14
D2H Transfer	2
Total	601

Transfer data is minimal: transfer from host to device is 5 ms, and transfer from device to host is 2 ms. Thus, once the traversal is moved to GPU, our main bottleneck is the CPU BVH construction as shown is the figure below:

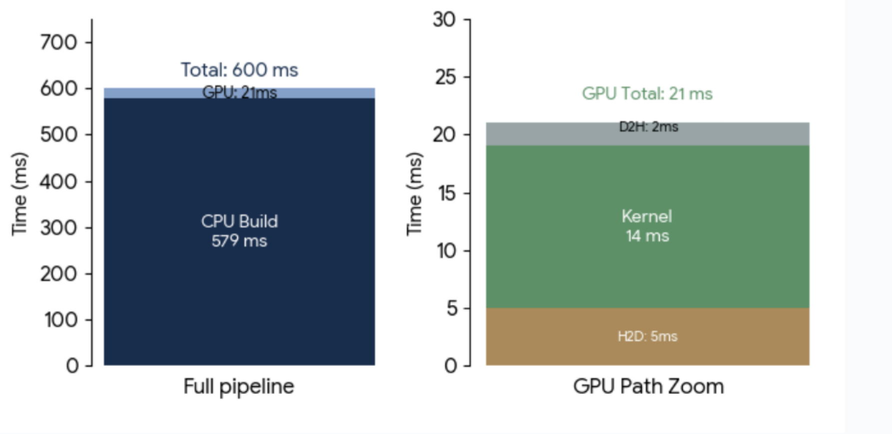


Figure 11: Hybrid Pipeline time breakdown for 4K dragon

Effect of Problem Size / Scene Complexity

Finally, we considered the impact of scene size on the parallel speedup. The reason why it is important is that there are certain overheads associated with building a BVH using OpenMP task parallelism, including OpenMP tasks' spawning, synchronization, partitioning, and memory management. With small scenes, the overheads overshadow the computational gain, and there is no sufficient amount of subtrees to engage multiple CPU cores. However, with bigger triangles, the hierarchical BVH structure consists of significantly more independent subtrees, and task parallelism becomes increasingly beneficial.

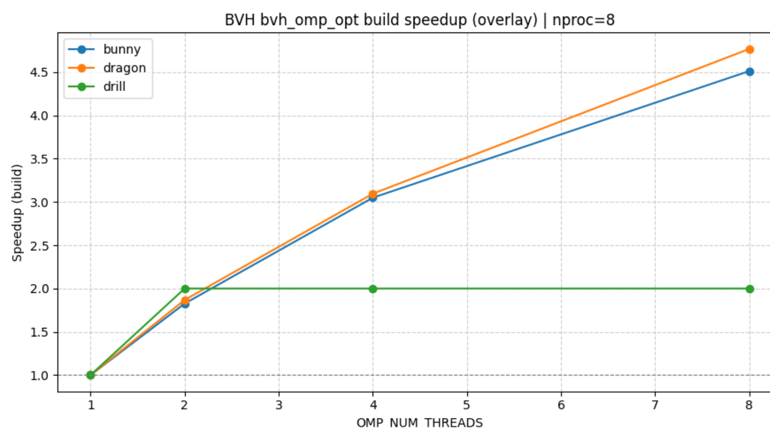


Figure 12: Speedup across different scenes

The pattern can be observed for Drill, Bunny, and Dragon. Specifically, Drill is too small to leverage the advantages of task parallelism, Bunny yields modest gains, and Dragon demonstrates considerable parallel potential, resulting in the most significant BVH construction speedup among the three datasets.

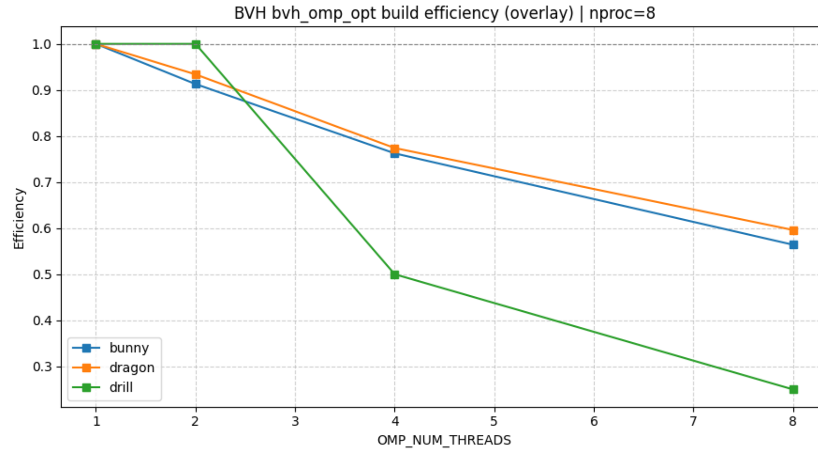


Figure 13: Build efficiency across different scenes

The results show that parallel BVH construction benefits more from larger scenes. In case of small examples like Drill, there is heavy overhead due to OpenMP and lack of parallelism of the sub-trees. Bunny example has sufficient number of primitives that can give rise to some amount of parallelism, but Dragon offers best scaling behavior because Dragon has hundreds of thousands of triangles which give rise to many sub-trees.

OTHER RESULTS

Frontier Hybrid Tradeoff

The hybrid frontier construction experiment demonstrates the trade-off between CPU BVH construction and the GPU leaf intersection processing. With an increasing number of frontiers, the CPU building time reduces as there is less work to do on the CPU in terms of BVH construction as shown in the table below:

Frontier Size	CPU Build	Kernel	Total
4	669	14	690
64	558	27	590
256	519	61	585
512	499	98	602
1024	476	159	641
2048	479	445	931
4096	464	724	1195
8192	442	1266	1715

On the other hand, the GPU kernel execution time increases as there are more triangles in each frontier leaf that must be intersected directly. The optimal range of frontier numbers would be from 64 to 256 primitive numbers per frontier leaf and shown in the graph below:

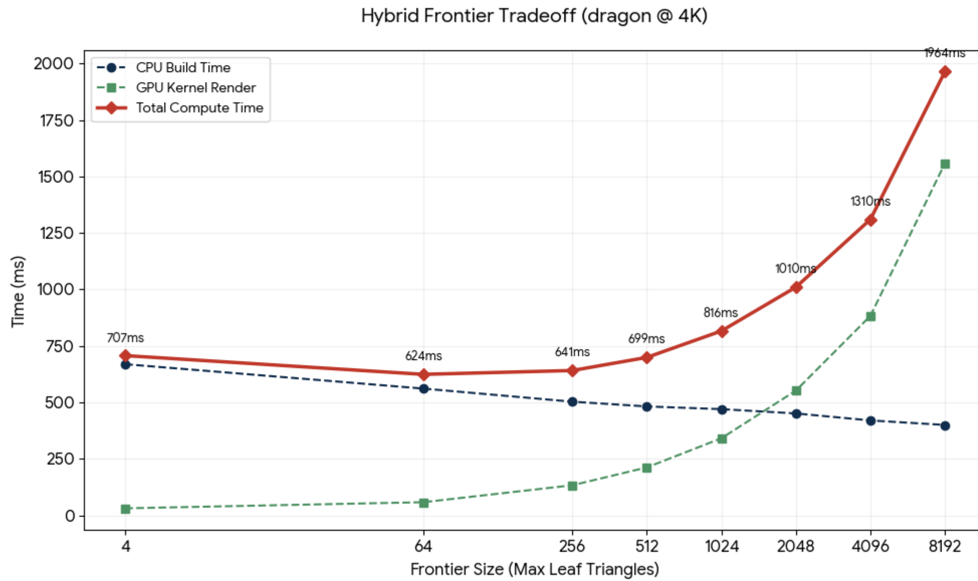


Figure 14: Hybrid Frontier Tradeoff for Dragon Scene at 4K

GPU Profiling

This was NOT run on the GHC machines, as we faced trouble getting Nsight Systems and Compute to work. These are run on a AMD Ryzen 7600x (6c12t) + RTX 3080 Desktop.

Profiling was performed at the default 800x450.

The trace results show that in case of hybrid pipeline, wall clock time is occupied mostly by CPU BVH generation. cudaMalloc allocation time (~96 ms, a single-time CUDA initialization cost) is second and kernel plus memcpy takes <5 ms. Table below shows Nsight Compute analysis results of hybrid_render_kernel on Dragon.

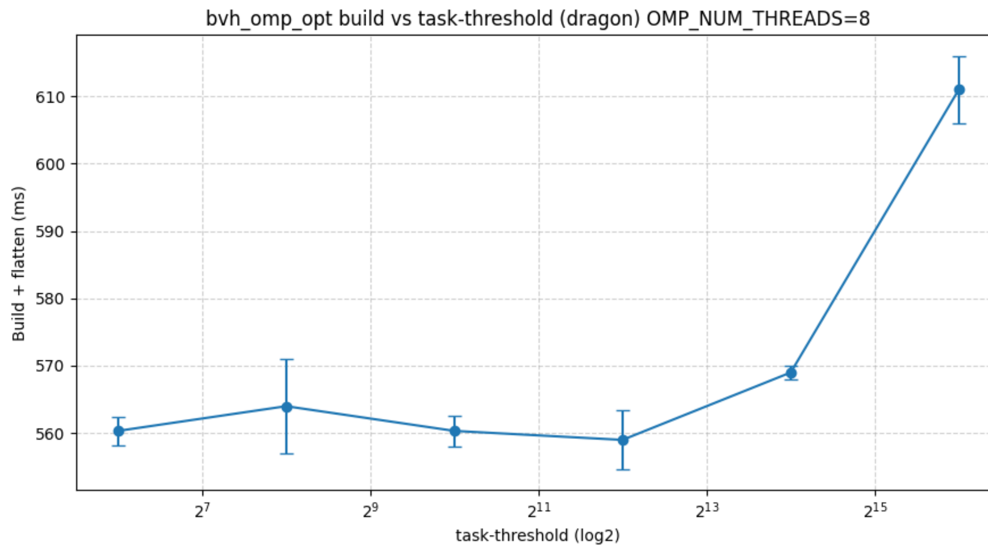
Metric	*bvh_cpu_gpu*	*bvh_cpu_gpu_flat*
Active threads / warp (max 32)	10.62	10.62
SIMT efficiency	33.20%	33.20%
Achieved occupancy	25.10%	25.70%
L1 hit rate	87.20%	83.30%
L2 hit rate	61.70%	52.90%
DRAM throughput (% of peak)	10.50%	13.60%
Kernel time (real)	421 µs	470 µs

This number corresponds to the amount of warp divergence predicted in the Locality subsection: on average only about one third of the threads in the SIMT lanes perform useful computations per cycle since nearby pixels traverse the BVH differently and terminate in different leaves. The achieved occupancy is about 25% (as opposed to the maximum possible 100%) due to register usage per thread in the local traversal stack. Despite this, the data throughput remains 10-14% of the peak value meaning that the kernel cannot saturate either the memory bandwidth or computational resources it is latency-limited by warp serialization.

An interesting observation was made regarding the impact of the flat layout format on the GPU caches: despite expectations the non-flat layout shows similar performance in terms of kernel execution time and even demonstrates higher hit rates in L1/L2 caches. Therefore, the improvement of the overall kernel execution time (see the Hybrid Performance Breakdown subsection) is solely attributed to the absence of CPU-side flatten operation.

Task Threshold

We tried all possible values for the OpenMP -task-threshold option that regulates how narrow the range of primitives should become before recursion tasks are stopped and serial recursion is performed instead. The build time on the Dragon does not change for almost two orders of magnitude (from 64 to 16,384), while a slight degradation (by about 9%) occurs only at 65,536.



This came as an unexpected observation as we expected finer tasks (64) to underperform due to the overhead incurred from the OpenMP runtime while very coarse tasks (65,536) to underperform due to serial tail performance. The flat profile shows that OpenMP runs well with this number of tasks as the work done by these tasks depends only on a few larger subtrees regardless of the chosen threshold.

What limited speedup?

The primary limiting factor for speedup in CPU BVH construction is that a top-down algorithm has many serial steps. A node must calculate its bounding volume and partition the primitives before its children are constructed. In addition, there is som extra work needed to partition the primitives and move data around. In our hybrid approach, the speedup is now no longer constrained by traversal. CUDA traversal is just 14 ms on 4K. The problem now becomes CPU BVH construction, which is respnsible for around 97% of the total hybrid computation. Moving the data is not the limiting factor here, as H2D and D2H operations are just about 7 ms on 4K.

Was the machine target sound?

The CPU/GPU selection was justified because of the difference between both stages regarding parallelism. Task parallelism in CPU is more appropriate for BVH construction due to its recursive nature, whereas traersal can benefit from GPU implementation as rays are independent. This is also demonstrated by the performance comprison: the CPU version of construction provided a useful but speedup of 4.71x using 8 threads, while the CUDA version of traversal provided approximately 904x speedup over the CPU version.

OUTPUT IMAGES

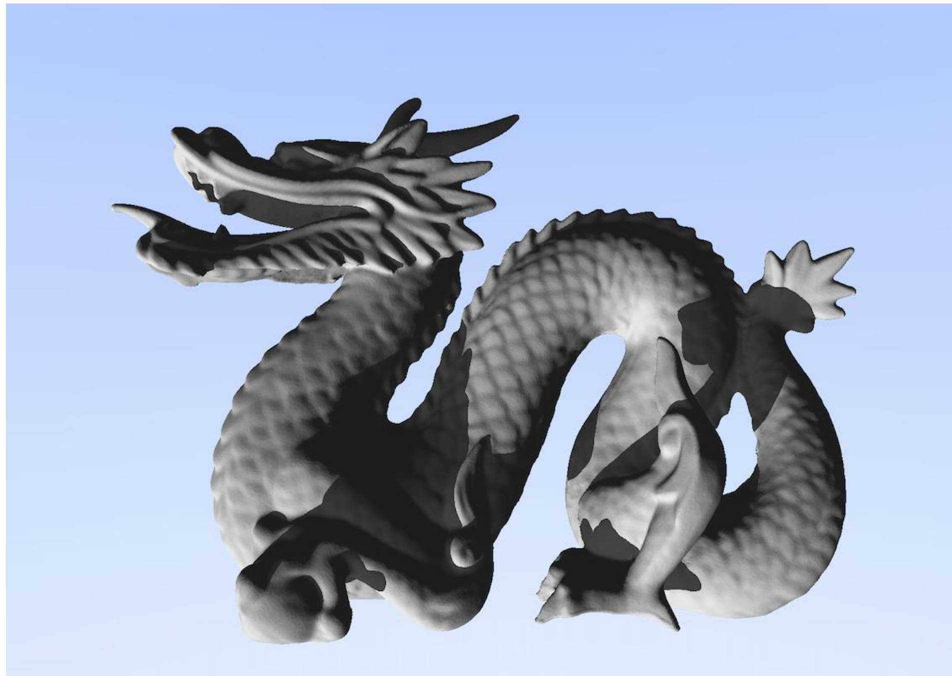


Figure: Dragon with lambert and shadows at 4K

Figure: Bunny at 4K

Figure: Drill at 4K with shadows

WORK DISTRIBUTION

Person	Contributions	Credit
Soham Narendra Joshi	CPU BVH construction, OpenMP optimization iterations, benchmarking infrastructure, hybrid CUDA integration, report and poster	50%
Viren Dodia	CUDA traversal kernel, GPU data structures and flat BVH, rendering integration, experiments, report and poster	50%

REFERENCES

[1] https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/

[2] PBRT BVH bucket/SAH partitioning discussion

[3] https://graphics.stanford.edu/data/3Dscanrep/

[4] A Volumetric Method for Building Complex Models from Range Images

[5] Peter Shirley. *Ray Tracing in One Weekend*.

[6] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in Proceedings of High Performance Graphics (HPG), 2012.

[7] T. Karras and T. Aila, "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies," in Proceedings of High Performance Graphics (HPG), 2013.