GPU Ray Tracer with Optimized Parallel BVHs

An Wu (anwu) & HingOn Miu (hmiu)

Table of Contents

1. Summary

We implemented a state-of-the-art GPU ray tracer with optimized parallel bounding volume hierarchies in CUDA. First, We built a parallel GPU ray tracer in CUDA. Then, we constructed parallel bounding volume hierarchies on GPU according to Tero Karras's Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. Finally, we optimized the bounding volume hierarchies according to Tero Karras's and Timo Aila's Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. Our CUDA implementation runs on NVIDIA's GRID K520 through Amazon Web Services.

2. Background

2.1 Overview

Ray tracing generates the colors of an image by tracing the path of the light rays for each pixel in an image. The light ray simulates realistic physical encounter with the objects along the path, including reflection and refraction. Hence, it is essential to compute the closest object that intersects with the light ray. So, the runtime of ray tracing effectively lies on the time spent on finding the closest intersected object among all geometries. Many researches are conducted aiming to find the most efficient data structure that reduces the geometries traversal time.

We parallelized construction of BVHs to accelerate ray tracing on GPU. Specifically, we took the CPU ray tracing code from smallpt that uses global illumination, and first parallelized it (per-pixel parallelism) to run on GPU. Then, we used the BVHs construction algorithm by Tero Karras, and BVHs optimization algorithm described by Tero Karras and Timo Aila to build the BVH used to accelerate ray tracing.

2.2 Data Structures & Operations

2.2.1 Radix Tree

The BVH tree construction involves 2 steps. The first is to construct a radix tree. It comes in the following form (Karras 2012):

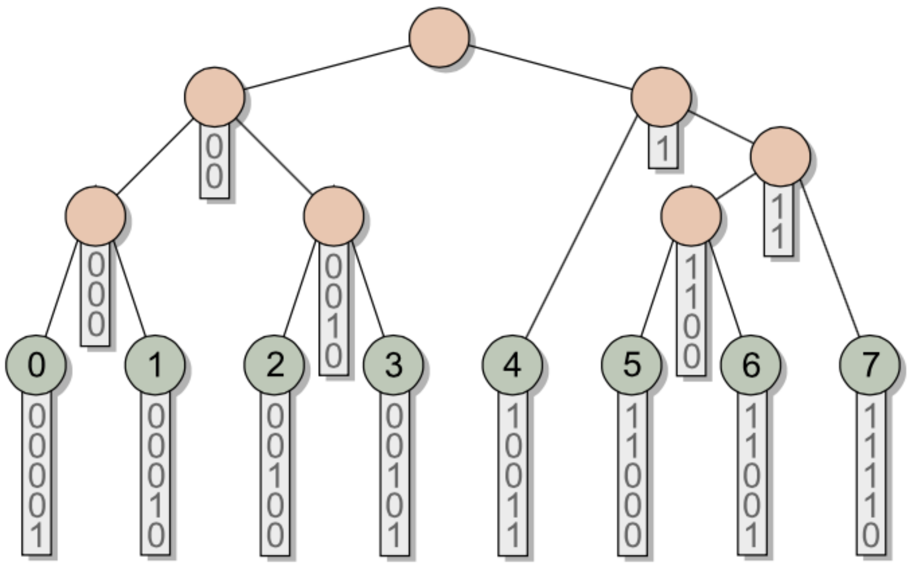


Figure 1: *Ordered binary radix tree. Leaf nodes, numbered 0-7, store a set of 5-bit keys in lexicographical order, and the internal nodes represent their common prefixes. Each internal node covers a linear range of keys, which it partitions into two subranges according to their first differing bit.*

The radix tree is as a balanced compact tree (every internal node has exactly two leaves) with a fixed number of leaves. We choose to construct a radix tree before the BVH tree because we can achieve per-node parallelism when constructing the radix tree. With a constructed radix tree, we can construct BVH in a bottom-up manner which we will explain in detail in section 2.3.

2.2.2 Bounding Volume Hierarchy (BVH) Tree

The second step is the actual BVH tree construction. The tree construction begins from bounding boxes of geometries in tree leaves, and then it merges the bounds going up to the root.

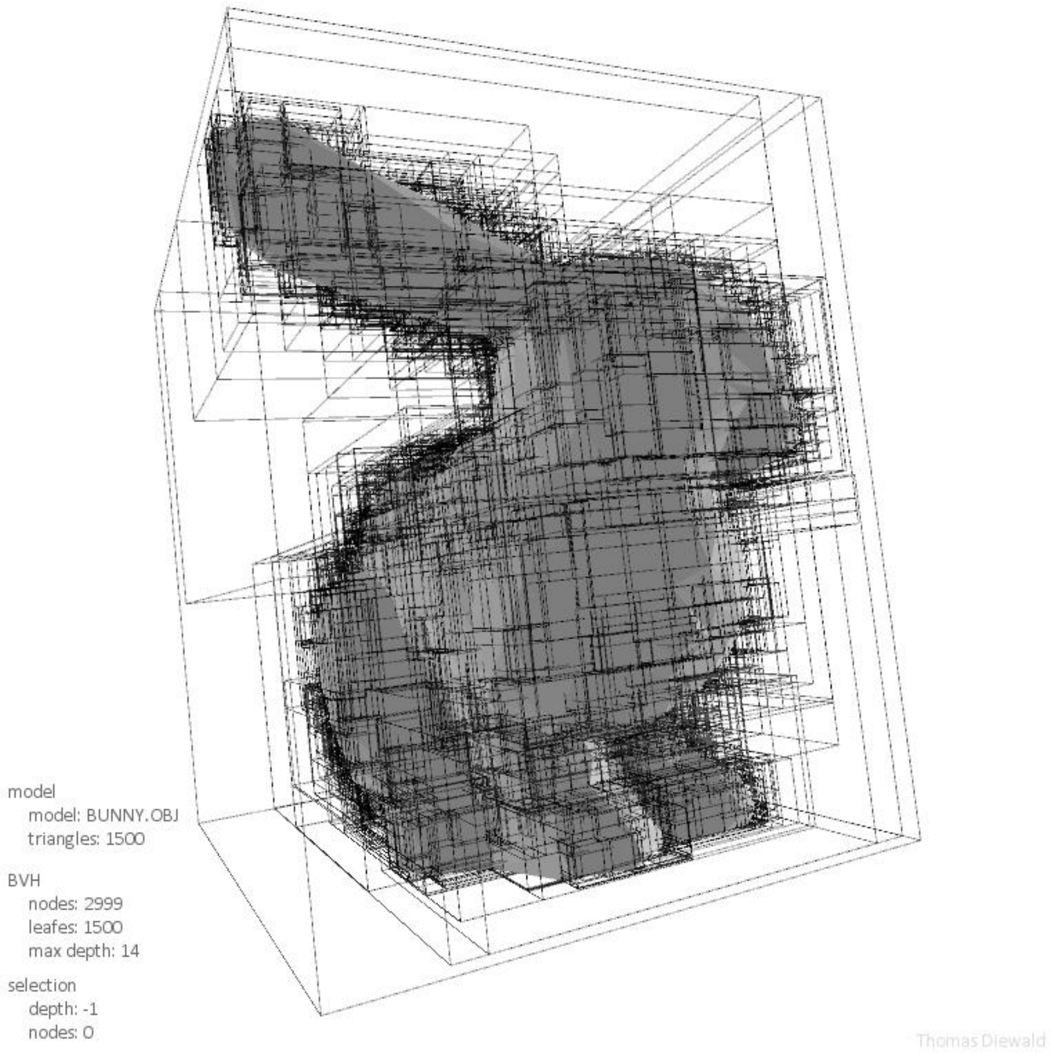


Figure 2: *A bunny represented in a BVH Tree.*
Image credit

Instead of traversing all geometries to test for intersection during ray tracing, we only need to visit a BVH tree node if the ray intersects its bounding box. This effectively decreases the runtime cost to *O(log(n))* in a graph whose geometries are evenly distributed.

2.3 Algorithms Inputs & Outputs

2.3.1 Radix Tree Construction

Inputs: Number of leaves
Outputs: A balanced radix tree that contains exactly specified number of leaves

2.3.2 BVH Tree Construction

Inputs: The constructed radix tree in section 2.3.1; a sorted list of geometries in the scene (see section 3.2 for sorting details)
Outputs: A BVH tree which has the same structure as the radix tree, and has its leaves being the geometries in order. All the tree nodes have merged bounding boxes from their children.

2.3.3 BVH Tree Optimization

Inputs: The BVH tree from section 2.3.2.
Outputs: An optimized BVH tree that's close to the golden standard of ray tracing.

2.4 Computations That Can Benefit From Parallelization

2.4.1 Ray Tracing using Global Illumination

The ray tracing in smallpt is implemented using global illumination. The ray tracing logic is enclosed in a per-pixel for loop. Due to the non-linear computation needed in ray intersection test as well as the multiple samplings needed to refine the image, the algorithm has high runtime cost. Ray tracing would benefit from simple per-pixel parallelization.

2.4.2 Accelerating Structure Construction

The ray tracing acceleration data structure (in our case, a BVH tree) construction wouldn't cost much, because most acceleration data structures construction time scale linearly regard to number of geometries. However, it could still benefit from parallelization because of its tree form.

2.4.3 Accelerating Structure Optimization

Depending on the optimization methods and the requirement of acceleration data structure quality, optimization could cost variably. In our case, optimization has linear cost with regard to geometry size (though with a relatively large constant multiplier). This could still benefit from parallelization because the optimization traverses the tree bottom-up, one level at a time.

2.5 Preliminary Analysis

2.5.1 Inter-structure

All data structures depend on the previous ones to be built: Optimized BVH Tree -> BVH Tree -> Radix Tree. This process cannot be parallelized.

2.5.2 Radix Tree Construction

The radix tree nodes are all independent of each other, so they can be per-node parallelized (see section 3.1 for details). It's fully data-parallel (high SIMD utilization), and locality is high because the tree nodes are kept in an array, and CUDA threads take nodes in order.

2.5.3 BVH Tree Construction

The BVH tree construction has dependency vertically. The levels on the top requires the levels beneath them to be built. Thus, we get a lot of parallelism at the bottom of the tree, and get less and less parallelism as we go up (same for data parallelism). However, at each level the locality is good, because nodes on the same level stay together in the memory. SIMD is not that good because at each level, we lose half of the CUDA threads.

2.5.4 BVH Tree Optimization Dependencies

The BVH Tree optimization process has the same structure of dependencies as the BVH Tree construction process on the high level, and thus the analysis of data parallelism, locality, and SIMD execution is very much alike.

However, the optimization dependency for individual treelet is different. With dynamic programming(see section 3.3.2 for details), the execution path is pre-determined, and thus we can get very high SIMD utilization. For every treelet, we frequently access 128 bytes of memory to find the optimal partition, and so the memory locality is high. However, when restructuring the treelet, nodes on different levels of the tree are accessed, which decreases data locality.

3. Approach

3.1 Radix Tree Construction

We can construct a balanced radix tree with any number of leaf nodes in a **per-node parallel** manner on GPU (as opposed to the usual bottom-up approach which slows down at the top parts of the tree). The approach can be demonstrated in the following graph (Karras 2012):

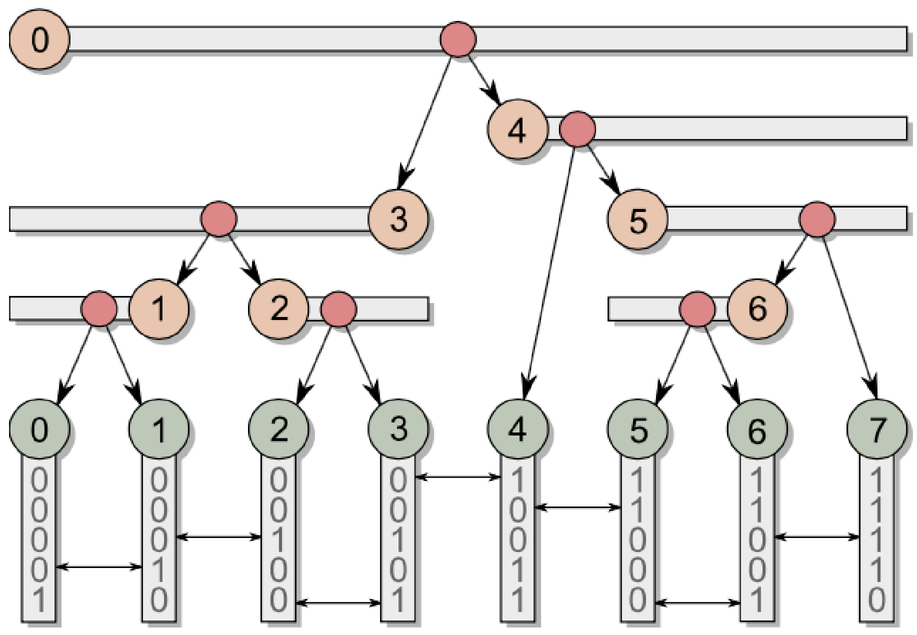


Figure 3: *Our node layout for the tree of Figure 1. Each internal node has been assigned an index between 0–6, and aligned horizontally with a leaf node of the same index. The range of keys covered by each node is indicated by a horizontal bar, and the split position, corresponding to the first bit that differs between the keys, is indicated by a red circle.*

Given the index i of a radix tree node (the number in the orange circles), we can find its sibling by comparing the length of common prefixes of (i, i+1) and (i, i-1) and take the smaller one. Given the length of common prefix, we can calculate the other end of the current segment by doing binary search (until the length of common prefix decreases). When this is done, we can link the current node to its children by calculating the split position of the current node.

3.2 BVH tree Construction

Given the constructed radix tree, we can populate the BVH tree in a **bottom-up** manner. We basically create a CUDA thread for each leaf. When going up from leaf to root, the first-to-come thread would terminate, and the second-to-come thread will merge bounds at the node and keep going up to root. In this manner, we can ensure that both children of a node are processed when the node is going to be processed. Since each node is processed by exactly one CUDA thread, the time complexity is O(n).

To improve the quality of the BVH tree, we adopt the **Morton code** method described in the Karras 2012 paper. Basically, we give a geometry a 60-bit morton code (put in a 64-bit long) that takes the form of X0Y0Z0X1Y1Z1…, where X0X1X2… is its centroid's x-coordinate, and Y0Y1Y2... is its centroid's y-coordinate, and Z0Z1Z2… is its centroid's z-coordinate. Then, we sort the geometries according to their Morton codes, and assign them to the leaves from left to right. In this way, closely positioned geometries are put together.

3.3 BVH Optimization

3.3.1 Treelet Rotation

Even though linear BVH construction described in section 2.3.1 and 2.3.2 is fast, their ray tracing performance tends to be unaccepted slow --- usually around 50% of the gold standard (Karras & Aila, 2013). Thus, we adopted the ***treelet* reconstruction** method described by Karras and Aila to improve the linear BVH tree to one that is close to the gold standard in ray tracing performance.

A *treelet* is defined as the collection of immediate descendants of a given treelet root, consisting of n treelet leaves and n-1 treelet internal nodes. Given a treelet root, we populate the entire treelet by recursively adding children of the node that has the greatest *Surface Area Heuristic* (SAH) cost. The **SAH cost** of a given acceleration structure is defined as the expected cost of tracing a non-terminating random ray through the scene. With a treelet, we can perform reconstruction on it to achieve the best SAH cost for any treelet node.

We choose treelets of size 7 in our implementation, because individual treelets provide us **more than ten thousands of ways of reconstruction** (Karras & Aila, 2013).

To summarize, we start from the leaves of the tree, form treelets while we go up, and rotate them to optimize the BVH's performance.

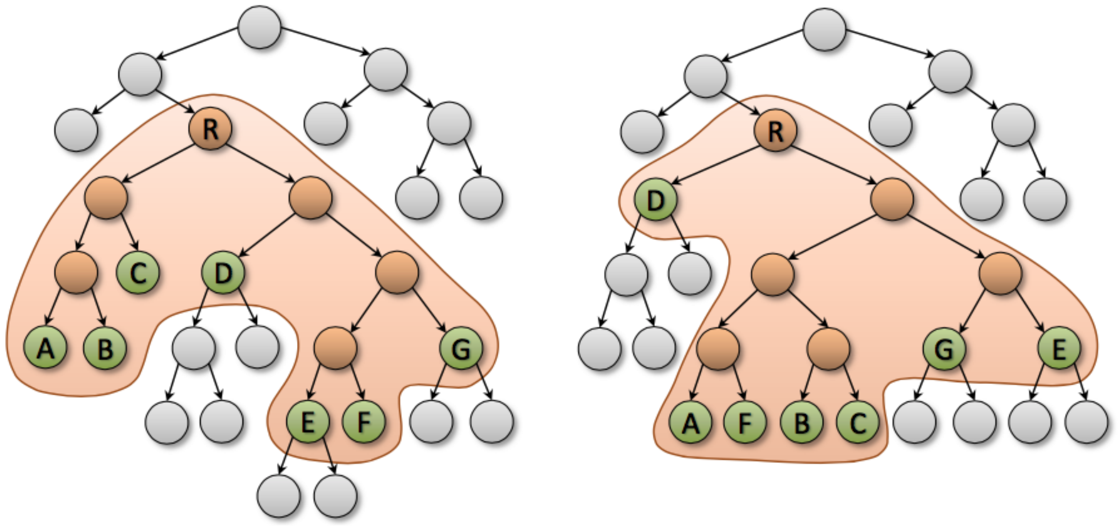


Figure 4: ***Left****: Treelet consisting of 7 leaves (A–G) and 6 internal nodes, including the root (R). The leaves can either be actual leaf nodes of the BVH (A, B, C, F), or they can represent arbitrary subtrees (D, E, G).* ***Right****: Reorganized treelet topology to minimize the overall SAH cost. Descendants of the treelet leaves are kept intact, but their location in the tree is allowed to change.*

3.3.2 Dynamic Programming

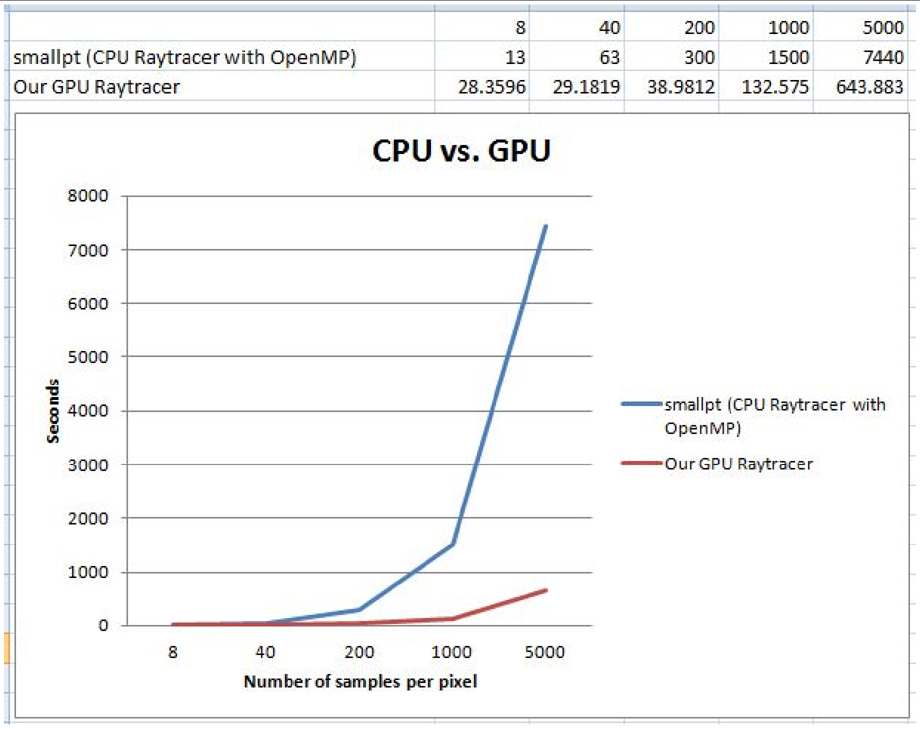
We acquire the best treelet rotation by recursively finding the best partition of left and right children of a node, until the partition size becomes one. Though this algorithm seems simple, it involves a lot of recursive calls and repeated work. Thus, we adopt Karras and Aila's dynamic programming approach to memoize the best results of the small partitions and use them to build larger and larger partitions. This way, we effectively transform from **a naive algorithm that makes 1.15 million (Karras & Aila, 2013) recursive calls** to **a single iterative function that has time complexity of *O(n)***.

Also, we use char bitmasks to indicate which treelet leaves are in the partition, since the size of the treelet is 7 (which is less than 8 bits, the size of 1 char).

4. Results

Graph 1: Runtime comparison between a CPU ray tracer and a GPU ray tracer

Constant	The sample scene contains 9 spheres
Dependent Variable	The runtime of ray tracing measured in unit of seconds
Independent Variable	# of samples per pixel = # of ray tracing execution per pixel
Configuration of blue line	Ray tracing is parallelized with OpenMP on CPU
Configuration of red line	Ray tracing is parallelized with CUDA threads on GPU



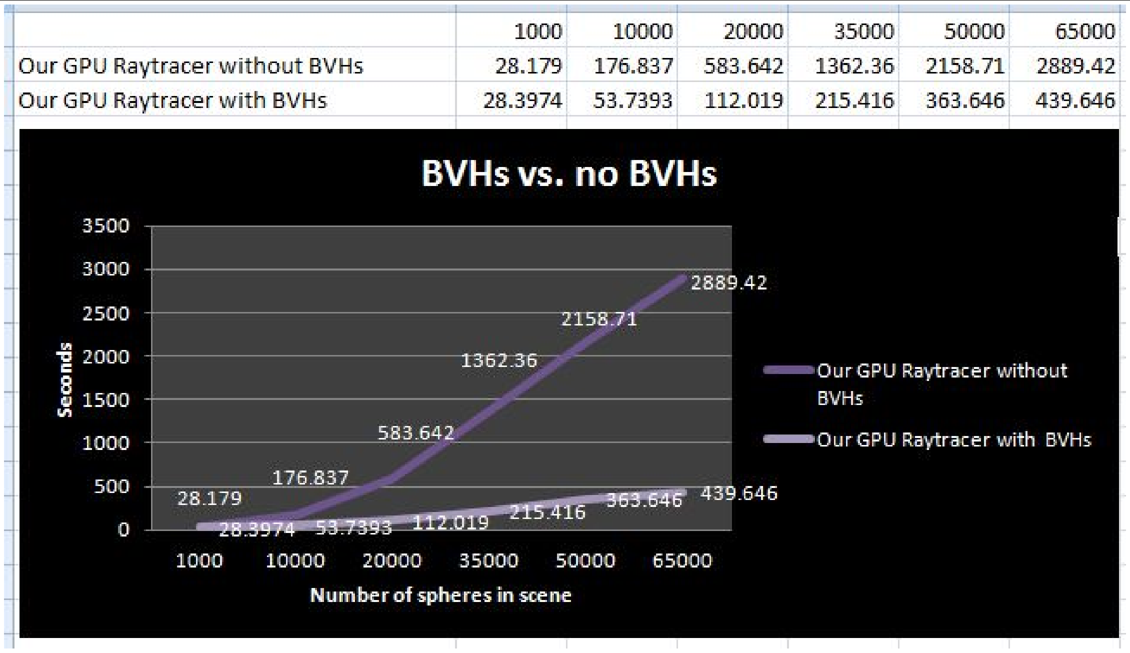
	8	40	200	1000	5000
smallpt (CPU Raytracer with OpenMP)	13	63	300	1500	7440
Our GPU Raytracer	28.3596	29.1819	38.9812	132.575	643.883

Problem Size Analysis	We choose to vary the number of samples per pixel because it defines the number of times of ray tracing computation. Naturally, we want to see the difference in time spent on ray tracing on CPU and GPU.
Runtime Analysis	This is a good runtime comparison of parallelism between CPU and GPU because the exact same piece of code are parallelized over CPU and GPU. The intersection tests of both implementation traverse all geometries sequentially, and so we can actually evaluate how the same piece of code is executed in different settings. Obviously, GPU implementation much faster. Notice that smaller samples per pixel have approximately 28 seconds GPU runtime, and so we speculate that this should be the overhead for launching CUDA threads.

	Since the geometries are traversed in for-loop in both implementation, there is memory locality for accessing subsequent closely stored geometries. Since both GPU and CPU implementations traverse all geometries sequentially, each geometry is fetched from memory to run intersection test. Since the size of the cache in CPU and GPU is limited and the memory bandwidth is finite, the runtime of finding the closest interested object lies on how fast to fetch each geometry from memory. Hence, we speculate that ray tracing is a bandwidth bound algorithm. For our CUDA implementation, it runs on a GPU with 16-wide SIMD. So, while each ray is parallelized, a set of 16 rays are computed concurrently. Since each ray can get absorbed, reflected, or refracted, a current ray can terminate, generate a new ray, or generate two new rays respectively. Therefore, we speculate the SIMD utilization is quite poor due to divergence since some lane in SIMD computing an absorbed ray must remain idle while some other lane computing multiple refracted rays.

Graph 2: Runtime comparison of GPU ray tracer with and without BVHs

Constant	4 samples per pixel = 4 execution of ray tracing per pixel
Dependent Variable	The runtime of ray tracing measured in unit of seconds
Independent Variable	The number of spheres in scene
Configuration of Purple line	Ray tracing is parallelized with CUDA threads on GPU. The geometries in scene are traversed sequentially in each intersection test.
Configuration of Grey line	Ray tracing is parallelized with CUDA threads on GPU. The geometries in scene are stored in BVHs, and the tree is traversed in each intersection test.



	1000	10000	20000	35000	50000	65000
Our GPU Raytracer without BVHs	28.179	176.837	583.642	1362.36	2158.71	2889.42
Our GPU Raytracer with BVHs	28.3974	53.7393	112.019	215.416	363.646	439.646

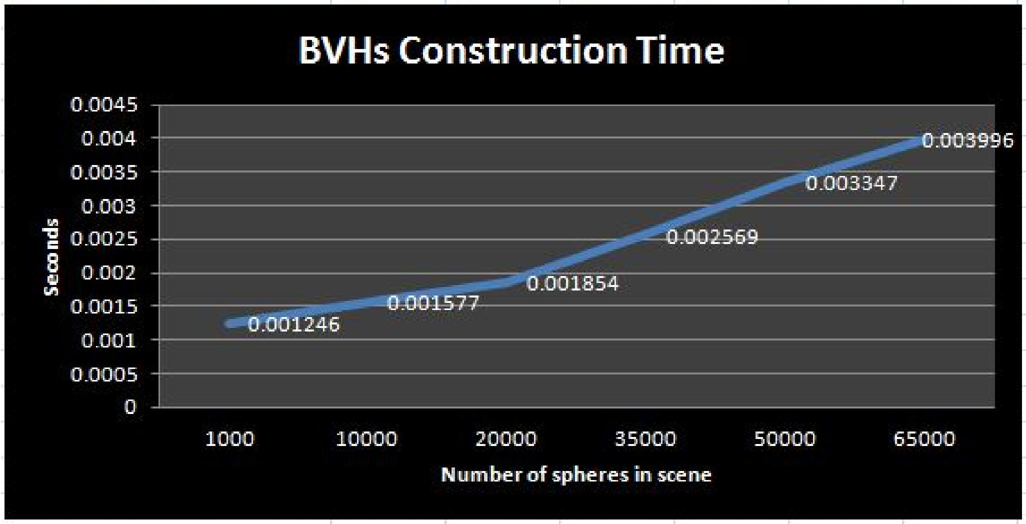
Problem Size Analysis	We choose to vary the number of spheres in scene because it defines the number of geometries to traverse in intersection test. Naturally, we want to see if the use of BVHs reduces the time spent on traversal.
Runtime Analysis	The actual ray bouncing computation is inherently sequentially, since there is no way to compute the next ray before the current ray intersects an object. Therefore, this ray bouncing computation is the limiting factor in parallelizing ray tracing in both CPU and GPU. Hence, it is essential to compute intersected object for each ray efficiently. Obviously, without BVHs, the runtime of finding the closest intersected object is O(n), since all geometries have to be traversed. Then, the best case with BVHs in runtime is O(log n), since exactly one branch of the tree is traversed so that the number of geometries accessed is bounded by the depth of the tree. However, the worst case with BVHs in runtime is still O(n), since both

	branches of each node are traversed so that all geometries are accessed anyways. Naturally, we require GPU to provide extra memory to store the BVHs. Therefore, this can potentially increase the runtime of ray tracing because the algorithm is bandwidth bounded. The memory access patterns of the sequential traversal and BVHs traversal of geometries are vastly different. The sequential traversal always access the geometries in some sort of fixed meaningless pattern. So, this exploits memory spatial locality because contiguously stored geometries are accessed. However, the BVHs traversal access the geometries in a spatial pattern. In other words, geometries that clustered in the same region are more likely to get accessed together. Hence, this exploits memory temporal locality because rays that shoot in similar directions are highly likely to access the same set of geometries in the BVHs. Therefore, we attempt to optimize the BVHs to further reduce the unnecessary memory accesses.

Graph 2.5: Correlation analysis of runtime of BVHs construction and # of spheres in scene

Dependent Variable	The time of BVHs construction measured in unit of seconds
Independent Variable	The number of spheres in scene
Configuration of Blue line	A radix tree is constructed parallelly on GPU, and then the geometries are inserted parallelly on GPU to construct BVHs.

	1000	10000	20000	35000	50000	65000
BVHs Construction Time	0.001246	0.001577	0.001854	0.002569	0.003347	0.003996

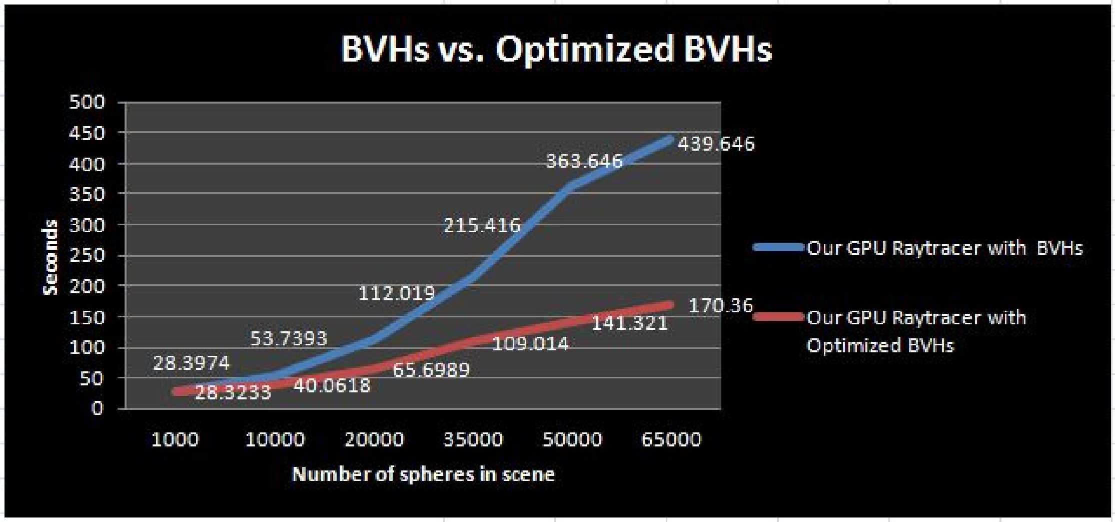


Problem Size Analysis	We choose to vary the number of spheres in scene because it defines the number of geometries to insert in BVHs construction. Naturally, we want to see if increase in geometries affects BVHs construction time.
Runtime Analysis	The BVH construction is composed of 2 parts: the radix tree construction, and the BVH tree construction. Since radix tree construction is done using per-node parallelism, it costs us little time compared to BVH tree construction, which uses a per-level parallelism. The total runtime is still linear with regard to number of geometries, because ultimately the number of threads will be much larger than the number of CUDA threads that can be run together. The radix tree construction has high SIMD utilization, because the same operations are done on each node in a data parallel fashion. The only conditional branch is at the end of the function, which is used to decide whether the child is another internal node or a leaf. It also has high data locality because consecutive indexed nodes are usually placed at the same level. Thus, CUDA threads in a warp usually access the same chunk of memory. The BVHs construction has relatively low SIMD utilization, because half of the CUDA threads terminate in each level of BVHs. The data locality is poor since the geometries are sorted, and so the memory access pattern is random in memory. When the CUDA threads process internal nodes, data locality rises because consecutively indexed nodes are placed in same level of BVHs.

Graph 3: Runtime comparison of GPU ray tracer with BVHs and with optimized BVHs

Constant	4 samples per pixel = 4 execution of ray tracing per pixel
Dependent Variable	The runtime of ray tracing measured in unit of seconds
Independent Variable	The number of spheres in scene
Configuration of Blue line	Ray tracing is parallelized with CUDA threads on GPU. The geometries in scene are stored in BVHs, and the tree is traversed in each intersection test.
Configuration of Red line	Ray tracing is parallelized with CUDA threads on GPU. The geometries in scene are stored in BVHs, and then the tree is optimized before ray tracing. The optimized tree is traversed in each intersection test.

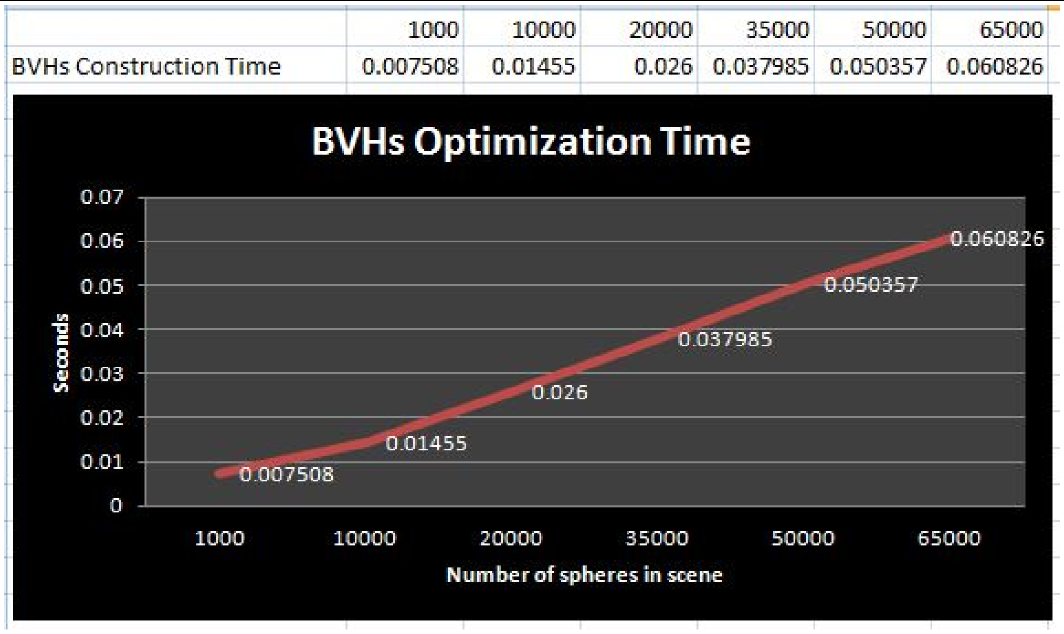
	1000	10000	20000	35000	50000	65000
Our GPU Raytracer with BVHs	28.3974	53.7393	112.019	215.416	363.646	439.646
Our GPU Raytracer with Optimized BVHs	28.3233	40.0618	65.6989	109.014	141.321	170.36



Problem Size Analysis	We choose to vary the number of spheres in scene because it defines the number of geometries to traverse in intersection test. Naturally, we want to see if the optimized BVHs is indeed faster.
Runtime Analysis	The ray tracing runtime with optimized BVH is much better than that of original BVH because the tree quality becomes much better. Hence, less branches are traversed such that the intersection test becomes more cache-friendly (less memory accessed). Obviously, we observe that the ray tracing runtime with optimized BVHs is more like O(log n). The SIMD utilization doesn't change much because the rays are still bouncing back and forward in random unexpected patterns.

Graph 3.5: Correlation analysis of runtime of BVHs optimization and # of spheres in scene

Dependent Variable	The time of BVHs optimization measured in unit of seconds
Independent Variable	The number of spheres in scene
Configuration of Red line	BVHs was already constructed on GPU, and then the tree is optimized (reconstructed) parallelly on GPU.



	1000	10000	20000	35000	50000	65000
BVHs Construction Time	0.007508	0.01455	0.026	0.037985	0.050357	0.060826

Proble m Size Analysis	We choose to vary the number of spheres in scene because it defines the number of geometries to insert in BVHs construction. Naturally, we want to see if increase in geometries affects BVHs construction time.
Runtim e Analysis	The runtime of BVH optimization scales linearly just as that of BVH construction, because they both operate in a bottom-up fashion such that half of the CUDA threads terminate in every level of BVHs. As a result, the graph is also linear with regard to the number of geometries. In the process of optimization, we used dynamic programming to figure out the optimal partitioning. Therefore, the data locality and SIMD utilization are both high. Data locality is high because our working set is an array of 2^7=128 characters (the total number of partitions of 7 leaves in a treelet is 2^7). SIMD utilization is high because the execution path in dynamic programming are pre-determined. Thus, even though our optimization brings the BVHs to a golden standard, the optimization step only cost a relatively little time.

5. Reference

I. Beason, Kevin. "Smallpt: Global Illumination in 99 Lines of C++." Kevinbeason.com. N.p., n.d. Web. 10 May 2015. <http://www.kevinbeason.com/smallpt/>.

II. Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics (EGGH-HPG'12), Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni (Eds.). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 33-37. DOI=10.2312/EGGH/HPG12/033-037 http://dx.doi.org/10.2312/EGGH/HPG12/033-037

III. Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In Proceedings of the 5th High-Performance Graphics Conference (HPG '13). ACM, New York, NY, USA, 89-99. DOI=10.1145/2492045.2492055 http://doi.acm.org/10.1145/2492045.2492055

6. Work Distribution

"equal work was performed by both project members."