High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Bandwidth-Efficient BVH Layout for Incremental Hardware Traversal

G. Liktor and K. Vaidyanathan

Intel Corporation

Abstract

The memory footprint of bounding volume hierarchies (BVHs) can be significantly reduced using incremental encoding, which enables the coarse quantization of bounding volumes. However, this compression alone does not necessarily yield a comparable improvement in memory bandwidth. While the bounding volumes of the BVH nodes can be aggressively quantized, the size of the child node pointers remains a significant overhead. Moreover, as BVH nodes become comparably small to practical cache line sizes, the BVH is cached less efficiently. In this paper we introduce a novel memory layout and node addressing scheme and map it to a system architecture for fixed-function ray traversal. We evaluate this scheme using an architecture simulator and demonstrate a significant reduction in memory bandwidth, compared to previous approaches.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Tree-based acceleration structures, such as kd-trees or BVHs (bounding volume hierarchies), are commonly used for the computation of ray-scene intersections. Being shallower than kd-trees and refittable under animation, BVHs dominate modern ray tracing. However, a typical BVH node stores a large amount of data, including 6 bounding planes and its child pointers.

The size of a BVH can be greatly reduced using quantization [Mah05] and parent-plane sharing [FD09, EW11]. Recently, Vaidyanathan et al. [VSAM16] combined these compression techniques with an incremental traversal algorithm that enables the use of reduced-precision arithmetic for the ray-box intersection tests, and can be implemented efficiently in fixed-function hardware.

Meanwhile, the memory bandwidth generated by BVH node fetches remains a significant problem, as we observe that the reduction of the bounding volume footprint does not yield a proportional improvement in bandwidth, partly due to poor cache locality. Yoon and Manocha [YM06] showed that cache locality can be substantially improved by reordering nodes in the BVH. However, their approach is not directly applicable to incrementally encoded BVHs, as it requires the storage of two child pointers, which yields a major overhead compared to the quantized node bounds.

We present a novel BVH memory layout for incremental traversal that improves cache locality and compresses the child pointers, in order to effectively reduce the bandwidth.

To this end, we make the following contributions:

- **A novel node layout and addressing scheme**
  We achieve memory bandwidth reduction at two levels: we compress the child pointers by forming clusters within the BVH, each within an arbitrarily reduced-precision address space. We then choose the order of nodes inside these clusters to maximize the cache line locality. We introduce a new node type to reference address-space changes during traversal. This keeps the node sizes uniform, which is more suited for a fixed function hardware.
- **Architecture model and bandwidth analysis**
  Building upon previous research in the area of fixed function ray traversal, we derive an architecture specialized for compact, incrementally encoded BVHs. Using a cycle-accurate simulation model, we compare our layout to existing popular layouts, including depth-first and clustered BVH with full-resolution pointers, and show an improvement of 15-35% in L2 to L1 bandwidth and up to 15% in L2 bandwidth under different workloads with varying ray coherence.

2. Related Work

We focus on the memory bandwidth of traversal and do not consider the other aspects of ray-traced rendering, like shading. We introduce the key factors that impact bandwidth, and discuss the previous work motivating our research.

DOI: 10.2312/hpg.20161192

BVH construction hierarchically partitions the primitives in a scene and stores a bounding box for each partition. The choice of these partitions has a major impact on the number of box and primitive intersections computed during ray traversal. A commonly used partitioning metric is the Surface Area Heuristic (SAH) [MB90]. Although not an accurate indicator of traversal performance [AKL13], it works well in most scenarios.

While construction quality directly affects bandwidth, our work is *orthogonal* to the topology of the BVH, only influencing its mapping to physical memory. In this paper, we build BVHs based on a binned SAH metric [Wal07].

Ray traversal on programmable architectures is a widely researched topic. Coherent rays can be grouped into small *packets* and traversed as SIMD batches [WSBW01]. However, many scenarios can produce rays with incoherent traversal paths. Other approaches use *multi-BVHs*, with several children per node that can be intersected together in a SIMD batch [WBB08, DHK08, EG08]. Although these are better suited for incoherent rays, the intersection tests for some children can be redundant. Coherence can also be extracted by sorting a large number of rays into batches that access the same set of nodes. The rays can be sorted per node [GR08], per multi-BVH node [BAM14, Tsa09] or per group of nodes [AK10]. Unfortunately, sorting can also generate a significant bandwidth just by the movement of rays.

Hardware ray traversal has not been as extensively researched as software implementations. Early hardware based architectures [SWW*04, WSS05] were also based on the SIMD processing of ray packets, and demonstrated the benefits of using a dedicated pipeline for traversing coherent rays. TRaX [SKKB09] and MIMD TM [KSBD10] introduced MIMD processing which was better suited for traversing incoherent rays. The T&I Engine [NPP*11] introduced dedicated hardware units for traversal and intersection, based on which Lee et al. [LSL*13] derived SGRT, an efficient ray tracing architecture based on BVH traversal for mobile GPUs.

The traversal stack is another source of memory bandwidth, which can be entirely eliminated with stackless traversal methods [KSS*13, fSK14]. However these rely on bidirectional pointers to backtrack traversal steps which precludes the sharing of bounding planes between parent and child nodes [FD09]. Therefore we opt for another approach by using a short on-chip stack and restarting traversal from the root node on a stack underflow [HSHH07, Lai10].

Incremental traversal is a recent technique to improve the efficiency of ray tracing hardware through low-cost reduced-precision arithmetic. Keely [Kee14] combined BVH quantization [Mah05] with a traversal method that incrementally translates the ray origin closer to the next BVH node. This allows performing the plane intersection tests with a reduced precision. Vaidyanathan et al. [VSAM16] improved the robustness of incremental traversal, and also further reduced the computational costs by reusing intersection tests from the parent nodes. This enables parent-plane sharing [FD09] in incremental traversal, which we utilize in our work.

2.1. BVH Memory Layout

Compact node ordering schemes can eliminate a few child pointers from the BVH. Depth-first layout (DFL) places the left child directly after the parent node, therefore only the the right pointer is required. Alternatively, two sibling nodes can be stored sequentially [AL09]. Besides compression, these layouts can also improve cache locality, since child nodes are often tested together following the parent during traversal. Nah et al. [NPK*10] improved cache locality using an *ordered depth-first layout* (ODFL), storing the child node with the largest surface area, next to the parent.

Subtree partitioning methods first decompose the BVH into clusters of nodes, each containing one or more subtrees. By optimizing the node order for multiple traversal paths it can further improve cache locality. Moreover, the size of the child pointers within clusters may be reduced. This optimization was presented for BSP trees by Havran [Hav97]. Gil and Itai [GI99] showed that cache locality for tree traversal can be significantly improved if the clusters of nodes are generated top-down, by greedily merging the children with the highest probability. Yoon et al. [YM06] applied this theory to kd-tree based ray traversal.

Aila and Karras [AK10] split the BVH into smaller treelets that are just large enough to fill the L1 cache. They limit the working set and therefore the bandwidth by scheduling rays that traverse the same treelet. Our algorithm on the other hand is designed for an architecture where the node sizes are small enough that the bandwidth of scheduling rays would become a significant overhead. It builds upon Yoon et al.'s clustering technique [YM06] but uses two levels of clustering to compress child pointers.

Havran's compact subtree layout removes pointers from internal nodes within subtrees, but the traversal algorithm needs to know the order of these nodes implicitly. In treelet clusters [AK10] nodes can reference children inside as well as outside the current treelet. Both of these solutions use heterogeneous node sizes, which would complicate node fetches in a hardware implementation. We unify node sizes by introducing of a new node type that allows indirections for child pointers. Furthermore, we also save some bandwidth by accessing these large pointers only when the child node is actually traversed.

3. Background

We begin the discussion of our memory layout optimizations with a brief formalization of the problem, following similar notations to previous work [GI99] [YM06]. Let $\mathbf{T} := \{BV_1, BV_2, ...BV_N\}$ be a *binary* BVH of N nodes. With the exception of the root, we can determine the parent of each node by defining the operator $parent(BV_i)$.

A descent $\mathbf{D}$ is a sequence of BV nodes $\{BV_{D_1=1}, BV_{D_2}, ..., BV_{D_k}\}$, where the first item is the root and $BV_{D_{i-1}} = parent(BV_{D_i})$; $i \in \{2...k\}$. We can also define a *ray query* $\mathbf{Q}$ as the sequence of nodes visited during traversal. It may visit the same node multiple times depending on restart operations. However, $\mathbf{Q}$ can be defined as a union of multiple descents, assuming that traversal starts from the root node.

By the *memory layout* of the BVH we mean a function τ that assigns a unique memory address to each BV_i. Following [GI99], we use the granularity of cache blocks, since the order within the same block does not influence bandwidth. Assuming uniform node sizes, a block can hold b nodes. We can define the layout as $\tau : N \rightarrow Z^+$, such that $|\tau^{-1}(i)| \leq b$. We define the *working set* of a descent as the number of different blocks accessed while iterating over its nodes:

$$WS_\tau(D^k) = |\{\tau(D_i^k)|1 \leq i \leq k\}|, \tag{1}$$

where D^k is the descend ending in the node BV_k. To reduce bandwidth, we seek τ^{opt} that minimizes the expected number of cache misses during traversal.

Gil and Itai [GI99] showed that finding τ^{opt} is NP-hard, but there are approximate solutions that seek to minimize the expected working set of a ray query, assuming that the probability of visiting each node, $Pr(BV_k)$, can be estimated:

$$E(WS_\tau) = \sum_{BV_k \in T} Pr(BV_k) WS_\tau(D^k). \tag{2}$$

The impact of memory layout on bandwidth is illustrated in Figure 1, where we show a simple traversal path accessing nodes through two different node orderings.

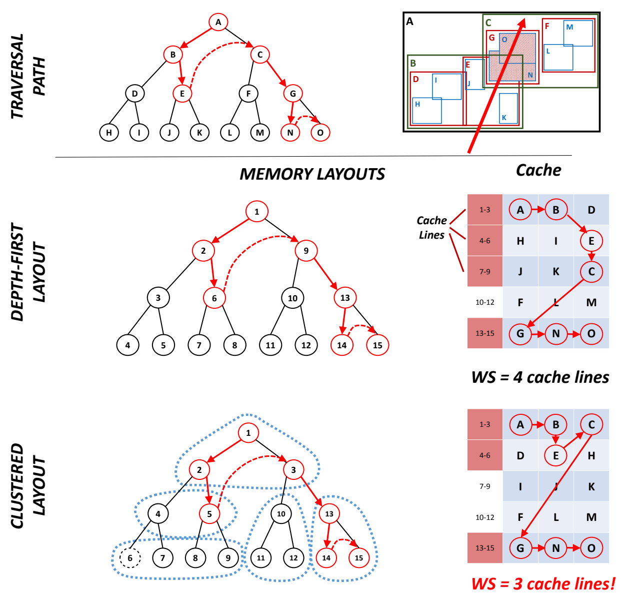


Figure 1: Traversal of a ray that intersects 2 leaf nodes (top). While it queries 7 nodes, more may be loaded due to the cache line size. Depth-first layout results in 4 transactions (middle), but clustering can reduce this to 3 (bottom).

4. Two-Level Clustering

When describing how changing the order of nodes can affect bandwidth, we *assumed that any τ layout fits into the same memory footprint*. In practice this may not hold. When using depth-first layout (DFL), an internal node can be represented as $\{P^{Right}|BV\}$: the pointer to the right child node, and the node bounds (BV). Since the left child directly follows, P^{Left} can be omitted.

We did not consider that storing P^{Left} allows less nodes per cache line. The size $|P|$ can be up to 4 bytes, which is small compared to conventional BVH encodings used by previous work. However, a node pair can be quantized to as small as 8 bytes using DFL (Sec. 6.1), whereas clustering would require up to 12 bytes. This 50% increase in node size counters the benefits of optimizing τ. **We therefore propose a two-level clustering scheme that allows node reordering while storing *two small* child pointers on the footprint of a regular pointer**:

- **Address Cluster (AC):** A continuous address space that can be referenced by a small pointer. If the original BVH can address 2^n nodes, the maximum size of an AC is $2^{n/2}$.
- **Cache Cluster (CC):** A small set of nodes that fits within a cache line, created within an *AC*.

The *AC* can maintain the node size of the depth-first layout (or even reduce it), while the *CC* reorders nodes within the same *AC* for the best cache utilization. Figure 2 provides a graphical overview of this hierarchical structure.

4.1. Glue Nodes

The use of small pointers limits the number of nodes within an *AC*. In order to support larger BVHs, we need a new node type that points outside this limited range. We call these "glue nodes" referring to their connecting role: they store a single full-precision pointer to the root of a new address cluster. The number of glue nodes is *much less* than internal nodes: if we assume that the original BVH contains $2N+1$ nodes (N internal), and the average size of an *AC* is $\sqrt{2N+1}$, the number of glue nodes is at the magnitude of $\sqrt{N}$. Using regular pointers would increase the size of at least N nodes. Furthermore, *glue nodes* only generate bandwidth when the child node is traversed, not when accessing the parent node.

5. Algorithm

In order to effectively reduce the working set, we must carefully select the nodes for each cluster. We adopt the probabilistic model proposed by Yoon and Manocha [YM06]. They attempt to order the nodes according to the most likely traversal path based on *parent-child* and *spatial locality*.

Assuming that all traversal paths start at the root of the tree, their *COLBVH (Cache-Oblivious Layout of BVHs)* algorithm iteratively merges the child nodes that are the most likely to be traversed next. *Parent-child locality* means the selection based on the conditional probability that the parent nodes in the cluster are already traversed. For ray tracing, the surface area is used as the probability measure. Once the cluster reaches its limit size, the remaining child nodes form the roots of new clusters and the process is repeated recursively until the entire tree is clustered.

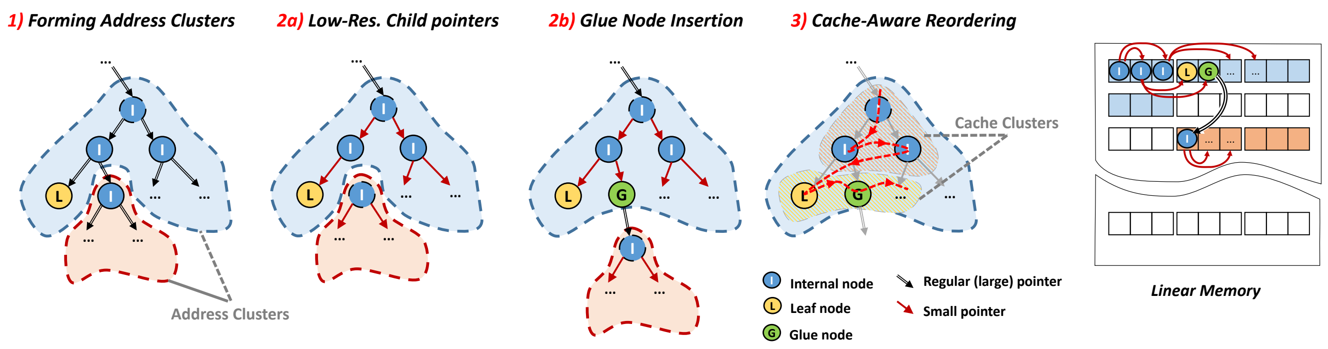


Figure 2: The key steps of our clustering algorithm. In the first pass the address clusters are created **(1)**, and connected by inserting glue nodes **(2a-b)**. This is followed by a conventional node reordering within each address cluster **(3)**. The rightmost figure illustrates the layout of address clusters in linear memory.

Spatial locality is used to define the order of clusters in linear memory. The authors provide two methods of different complexity to determine such an ordering: a graph-based approach inspired by a cache-oblivious mesh layout algorithm [YLPM05], and a simpler, but also faster multi-level depth-first layout.

Our algorithm is the modification of the COLBVH construction for two-level clustering. We have also made the method cache line aware, which allowed further important optimizations. The key steps of our algorithm, illustrated in Fig. 2, are the following:

0. Input: conventional BVH **T**, using e.g. depth-first layout; the desired child pointer size as **n** bits.
1. From the root of **T** recursively create address clusters (*AC*) using the COLBVH heuristics. An *AC* becomes full if the sum of its nodes and child clusters reaches 2^n. When an *AC* is full, the remaining children form the roots of new *ACs*.
2. For each child *AC* we create a *glue node* within the parent *AC*. Then step 1 is recursively repeated for each child.
3. Finally, within each AC we recursively create cache clusters (*CC*), again by using the COLBVH heuristics.

5.1. Address Clusters

Algorithm 1 lists the pseudocode for the address cluster construction. The function BUILDAC gets recursively called with a destination offset and root node (ln. **3, 23**). Once the address cluster is allocated, the function returns the updated value of `dstOffset` that points to the first available address for the next cluster (ln. **24**).

The first part of the method merges child nodes until the AC gets full, or there are no more child nodes remaining (ln. **6-11**). The function **pop_max_SA** refers to the selection of the child with the largest surface area. In the *AC* we also need to reserve a slot for its current children, since each of them is either a leaf node (which will be merged later) or an internal node. An internal node will either get merged, or a glue node is created that references it. Once this loop terminates, we merge the remaining nodes and create glue nodes for the child clusters. Note that the offset of the child cluster roots is not yet known, therefore these glue nodes must be updated later (ln. **22**).

With all the nodes in place, we can reorder them within the current *AC* using the COLBVH heuristics (ln. **20**): see Sec. 5.2 for details. We also ensure that the root of each *AC* is aligned with the cache line size, otherwise the cache-aware clustering step would not be effective (ln. **21**).

Algorithm 1 Recursive building of address clusters.

```
 1: ClusteredLayout($in\ srcBVH, out\ dstBVH$)
 2:     BuildAC(0, **get_root**(srcBVH));

 3: BuildAC($dstOffset, srcRoot$)
 4:     $maxN \leftarrow 2^{ptr\_bits}$; $AC \leftarrow \{\}$;
 5:     $childNodes \leftarrow \{srcRoot\}$;

 6:     while (size($AC + children$) < $maxN$ and !empty($children$))
 7:         $node \leftarrow$ pop_max_SA($childNodes$);
 8:         push_back($AC, node$);
 9:         if (is_internal($node$))
10:             insert($children, node.left$);
11:             insert($children, node.right$);

12:     $childACs \leftarrow \{\}$;                ▷ AC is full or no children left
13:     for all ($node \in childNodes$)
14:         if (is_internal($node$))
15:             push_back($childACs, node$);
16:             push_back($AC$, make_glue($node$));
17:         else
18:             push_back($AC, node$);

19:     $dstOffset$ +=BuildCCs($dstOffset, AC$);

20:     for all ($rootNode \in childACs$)
21:         $dstOffset \leftarrow$ cache_align($dstOffset, cacheLine$);
22:         update_parent_glue_node($rootNode$);
23:         $dstOffset \leftarrow$BuildAC($dstOffset, rootNode$);

24:     return $dstOffset$;
```

5.2. Cache Clusters

The creation of cache clusters is detailed in Alg. 2. Our clustering is *cache aware*, which yields two important differences compared to COLBVH. First, we do not need to recursively call the method within clusters, but define the size of the *CCs* to the number of nodes within a cache line. Second, this also allows us to better align CCs to cache line boundaries: full clusters that exactly match the line size can be moved to the beginning of the *AC*. We have found that if the line size is known, this optimization yields a significant reduction in the cache misses (Sec. 7).

Our construction is iterative, where the *active CCs* (clusters that are not full and have at least one child node to be merged) are kept in a sorted list (`CCRoots`). In each iteration we select the first active cluster and keep merging its children while it is possible (ln. **8-13**). If there are unmerged children left, we insert them *at the beginning* of the `CCRoots`. This defines the overall ordering of cache clusters. If the cluster is complete, but not full, we store it in the `deferredCCs` list, which gets written only after all full clusters are processed (ln. **19-20**).

The WriteCluster function finalizes the node addresses. An implementation challenge not detailed here is that when internal nodes are stored, the addresses of their children are not yet known. We need to maintain a look-up mechanism to retrieve the parent of a node and update its child pointers when their children are stored (ln. **23**).

The order of Cache Clusters Our *CCs* within each address space are stored in a **multi-level ordered depth-first layout**: after each cluster we store its children, before any sibling cluster would be listed. This is why we insert the list of child clusters at the beginning in line **14**. In contrast to [YM06], we have found that ordering these clusters based on their surface area (with the largest at the front) gives slightly better results than based on their position in the BVH. Therefore we keep the list `childNodes` sorted, so the operation **pop_max_SA** also simplifies to popping the head of the list.

Algorithm 2 The creation of cache clusters within an address cluster.

```
1:  BuildCCs($dstOffset, AC$)
2:      $maxN \leftarrow cacheLineSize/nodeSize$;                ▷ CC size
3:      $CCRoots \leftarrow$ get_root($AC$);                     ▷ List of CC root nodes
4:      $deferredCCs \leftarrow \{\}$;                          ▷ store CCs for late writing
5:      while (!empty($CCRoots$))
6:          $childNodes \leftarrow \{$pop_front($CCRoots$)$\}$;
7:          $CC \leftarrow \{\}$;
8:          while (size($CC$) < $maxN$ and !empty($childNodes$))
9:              $node \leftarrow$ pop_max_SA($childNodes$);
10:             push_back($CC, node$);
11:             if (is_internal($node$))
12:                 push_back($childNodes, node.left$);
13:                 push_back($childNodes, node.right$);
14:         $CCRoots \leftarrow \{childNodes + CCRoots\}$
15:         if (size($CC$) == $maxN$)
16:             $dstOffset \leftarrow$WriteCluster($dstOffset, CC$);
17:         else
18:             push_back($deferredCCs, CC$);
19:     for all ($CC \in deferredCCs$)
20:         $dstOffset \leftarrow$WriteCluster($dstOffset, CC$);
21:     return $dstOffset$;
22: WriteCluster($dstOffset, CC$)
23:     update_child_ptr(get_parent($CC[0]$));
24:     for ($i := 0; i <$ size($CC$); $i$++)
25:         $dstBVH[dstOffset] \leftarrow CC[i]$;
26:         $dstOffset$++;
27:     return $dstOffset$;
```

5.3. Further Optimizations

We experimented with further refinements of the algorithm that we omitted from the above pseudocode for brevity:

- **Padding:** a cluster is *fragmented* if it overlaps more cache lines. Traversing such a cluster may generate more cache misses. Alg. 2 already writes *full* clusters to the beginning of the address space. We found that instead of storing all remaining clusters directly after each other, a small amount of padding could be beneficial. E.g. if the cache line fits 8 nodes and 7 are already occupied, we align the next CC to the beginning of a new cache line (if it is larger than 1). We also need to limit padding, otherwise it would move nodes outside the range of the small pointers.
- **Cluster Merging:** Alg. 2 merges only child nodes of the current CC. We have experimented with two different heuristics to merge sibling nodes instead. The first one, also proposed in [YM06], merges a sibling if it overlaps the parent node with a larger surface area than the area of the child. Our second heuristic merges small completed clusters: by interleaving their nodes and sorting them according to surface area, we are likely to find larger nodes on the same cache line.

6. Architectural Simulation

In order to evaluate bandwidth with our memory layout, we derive an architecture that is suited for the reduced-precision traversal method of Vaidyanathan et al. [VSAM16]. We build upon the work of Lee et al. [LSL*13], which is focused on an energy-efficient architecture based on dedicated hardware for traversal and primitive intersection. Our analysis focuses on the performance of ray traversal and ignores the cost of shading, which is beyond the scope of this paper.

6.1. BVH Node Structure

Our compact node structure combines node quantization [Mah05] with parent plane reuse [FD09](Sec. 2.1). This allows us to represent a pair of sibling nodes by just storing 6 planes and a 6-bit reuse mask. The number of quantization bits defines an important tradeoff between memory footprint and BVH quality. With more aggressive quantization, the memory bandwidth of a single traversal step reduces, but also the bounds become increasingly conservative, which leads to more node overlaps and redundant intersection tests. In our system we use 6 bits per plane, because it allows us to store the entire structure on 8 bytes, as shown in Fig. 3. This is well-suited for practical cache line sizes, but increases the average traversal steps between 5-20%, depending on the BVH quality [VSAM16]. In Sec. 7.2 we briefly show results using different node sizes.

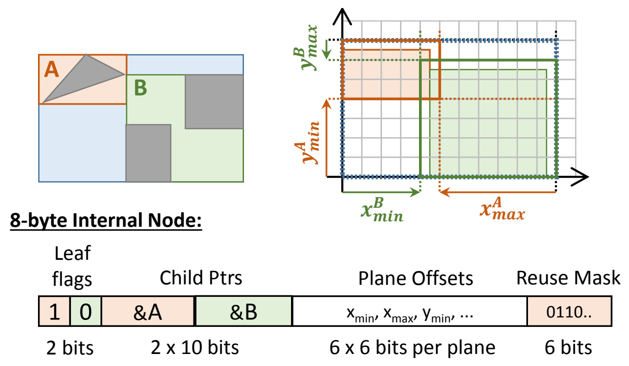


Figure 3: A 2D illustration of our quantized storage of sibling nodes with parent-plane sharing (top). The layout of our internal nodes (bottom). We store 2 bits to indicate leaves, one low-precision pointer per child, and 6 plane offsets (z-axis not shown). Finally, our reuse mask is set to 1 if the corresponding plane belongs to the left child.

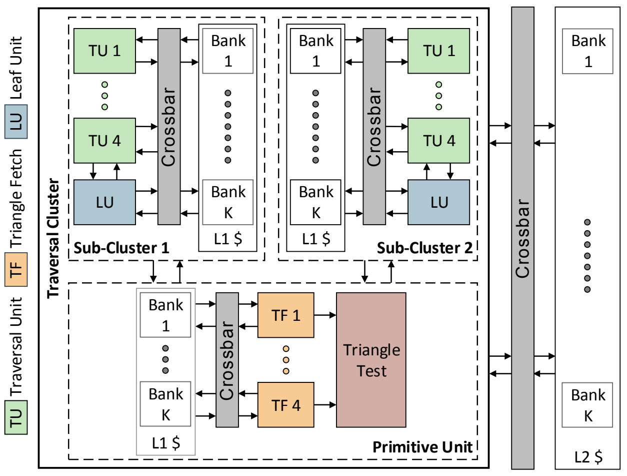


Figure 4: The block diagram of a Traversal Cluster

Jointly storing a pair of sibling nodes also moves the bounds of each node one level higher, which has important consequences. Most importantly, child nodes are only fetched on guaranteed intersection, eliminating redundant bandwidth. Second, since the leaf nodes contain no bounding plane data, their processing should be moved to a dedicated *leaf unit*. In order to know in advance if the next node to be traversed is a leaf, we reserve 2 bits per internal node for this information, and also assume that the root of the BVH is always an internal node.

6.2. Traversal Cluster

The **Traversal Cluster** is the primary building block of our architecture, which handles traversal and node intersections for several rays in parallel as shown in Fig. 4. It consists of several Traversal, Leaf and Primitive units, each of them pipelined and multi-threaded for out-of-order processing.

Traversal Unit (TU): traverses one ray per thread and stores its traversal state, which includes a short stack of 4 entries and a restart trail [Lai10]. We do not use the stackless traversal approach of Keely [Kee14] as it requires bidirectional pointers and prevents the sharing of bounding planes with the parent box. The TU can fetch and process one node pair every cycle. When a thread is ready to process a new node pair, a fetch request is sent to the L1 cache. On receiving the node pair, the TU pipeline decompresses the bounding boxes, computes ray-bounding box intersections and updates the traversal stack for the corresponding thread.

Leaf Unit (LU): fetches and processes leaf nodes as well as glue nodes. It includes a *node fetch* and a *dispatch* stage. The node fetch stage is identical to the TU. The node data includes a bit flag which indicates if a primitive leaf or a glue node was fetched. For primitive leaves, the LU issues a request to the PU for triangle intersections. If a glue node is received, the root offset of the new address cluster is sent back to the TU that initiated the request.

Primitive Unit (PU): performs the ray-triangle intersection tests. This can be an arbitrary full-floating point precision algorithm; our implementation is based on Woop [WBW13]. The vertex indices and position data for this test are fetched by *triangle fetch* units using 4 successive 12-byte accesses, backed by a separate L1 cache. The received vertex positions go through a pipelined *triangle test* unit, which sends the intersection results back to the requesting TU. We allocate 4 fetch units to fully utilize a single intersection pipeline.

6.3. Memory Hierarchy

With a compressed node structure of just 8 bytes, the bandwidth between the TU and the L1 cache is significantly reduced. In contrast to the T&I Engine of Lee et al. [LSL*13], which has a dedicated L1 cache per-unit, this allows us to share the L1 cache across several traversal and leaf units. We use a banked L1 cache and a crossbar network with a narrow 8-byte data width. When multiple TUs fetch nodes from the same L1 cache bank *bank conflicts* occur and the parallel accesses to the same bank are serialized. To avoid stalls, we address this problem by:

- introducing more banks than the number of T/L/P units, reducing the number of bank conflicts.
- introducing sufficient number of threads per unit to hide the latency of bank conflicts.
- each TU has a dedicated local storage (L0) for the top 3 levels of the BVH, since these nodes are often accessed.

We observe that the ratio of traversed internal nodes to triangles is greater than 8 : 1 for most of our test scenes. Therefore we introduce one PU and 8 TUs inside each traversal cluster. Since sharing the L1 cache across more than 4 TUs introduces significant latencies, we introduce a smaller logical unit called the **Sub-Cluster**, which assigns a group of 4 TUs to an 8-bank L1 cache.

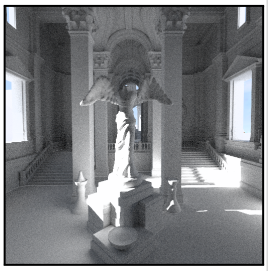
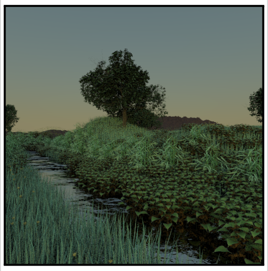
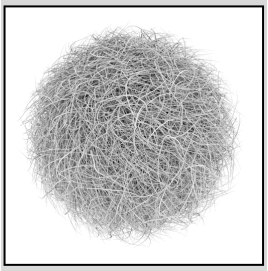
TEMPLE		CROWN		PLANTS		HAIRBALL		SANMIGUEL	
									
Tris	**Nodes / Ray**	**Tris**	**Nodes / Ray**	**Tris**	**Nodes / Ray**	**Tris**	**Nodes / Ray**	**Tris**	**Nodes / Ray**
500K	90I / 6L / 7G	5M	60I / 8L / 3G	1.2M	134I / 8L / 7G	2.5M	90I / 20L / 6G	1.6M	148I / 8L / 7G

Figure 5: The scenes used in our analysis. TEMPLE is rendered with diffuse one-bounce indirect lighting, while CROWN goes up to 4 bounces. PLANTS and HAIRBALL use environmental lighting. SANMIGUEL has direct lighting from a single light. We also show the average nodes accessed per ray (I-internal, L-leaf, G-glue), where glue nodes are only used in our BVH layout.

We ignore the latencies of the arithmetic pipeline in the TU and PU as it can vary with different bounding-box and primitive intersection techniques. Moreover, with sufficient threads to hide this latency we do not expect it to have a direct impact on memory bandwidth. The L1 caches fetch missing cache lines from a higher-latency L2 cache, which is banked and shared across all traversal clusters. This L2 cache could potentially be shared with the L2 cache on current GPUs. The L2 cache can fetch 64 bytes of data from higher levels in the memory hierarchy, such as a Last Level Cache (LLC) or DRAM. Although we specify a latency for these fetches, we do not model these higher levels, leaving this analysis for future work.

7. Results and Analysis

Based on the architecture described in Section 6, we simulate a configuration having 4 traversal clusters and a shared L2 cache, which can perform intersection tests for 32 node pairs per cycle (8 TUs in each cluster). Assuming an operating clock frequency of 1 GHz, we can achieve a ray throughput of approximately 200-350 Mrays/s for our test scenes with this configuration.

We analyzed performance statistics with a focus on two main aspects. First, we show how our node reordering algorithm reduces bandwidth compared to the conventional ordered depth-first layout. This is not only significant for energy-efficiency, but the reduction of cache misses may also increase the overall throughput. However, as we discuss in the second part of our analysis, low bandwidth by itself is not a guarantee of higher performance. To this end, we present some of the architectural challenges that need to be addressed for a scalable ray tracing architecture.

7.1. Workloads and System Configuration

We use 5 different test scenes in our evaluation within a wide range of geometric complexity, as well as material and lighting combinations, resulting in ray workloads with varying degrees of coherence. The TEMPLE scene has moderate geometric complexity with incoherent rays, while the CROWN scene has very high geometric complexity with more coherent rays resulting from a large number of glossy surfaces. PLANTS is an outdoor scene with incoherent rays sampling the environment light and fewer primitive intersections. HAIRBALL is a test scene with extreme geometric complexity and incoherent rays. SANMIGUEL is a scene with high geometric complexity and high ray coherence. Figure 5 shows the basic attributes of these workloads.

Since our simulation focuses on ray traversal only, we treat shading and ray generation as a "black box". We stream *traces* of rays captured from PBRT [PH10], that are distributed among our Traversal Clusters in tiles of 4×4 pixels. For CROWN we used the Intel Embree [WWB*14] renderer. Note that some of the scenes feature *instancing*, where the same BVH is traversed for multiple objects in the scenes, greatly improving cache-efficiency. We added an *idealized* instancing support to our system: instance nodes are special leaf nodes that are processed in the LUs. While our node and triangle bandwidth measurements are realistic, we do not account for the bandwidth generated by instance metadata (geometry headers and transformation).

Our system has several parameters, giving us some degree of freedom to scale hardware resources for the desired ray throughput. We have already discussed the number of T/L/P units that were based on typical leaf-to-internal node ratios in our test scenes. We model the cache hierarchy up to the level of the L2 cache, we set the hit latencies to $\{t_{L1} = 4; t_{L2} = 32; t_{LLC} = 100\}$ cycles, where LLC is the last level cache, and always produces a hit with a constant latency. We explore the performance of our system using varying L1 and L2 sizes, but unless otherwise stated, we set the $L1_{Node} = 40KB, L1_{Tri} = 32KB$ in each cluster (split evenly among subclusters), and an L2 size of 512 KB. We have found that for most workloads the resulting latency could be efficiently hidden using 16 threads on each TU and 32 threads on each LU and PU. The number of threads plays an important role not only in latency hiding but also out-of-order processing, since we do not rely on ray sorting, in constrast to [Kee14] or [AK10].

7.2. Bandwidth Analysis

L1 bandwidth In Fig. 8A we show the overall bandwidth requirements of traversing the same set of rays using different BVH node layouts. Using the ordered depth-first (ODFL) layout as a baseline, we show how node reordering using glue nodes (OURS) can reduce both the L1 and L2 cache bandwidth, in contrast to the cache aware variant of Yoon et al.'s method [YM06](CALBVH), which requires an additional 4 bytes per node for the second child pointer.

Using our method, we can observe more than 30% reduction in L1 node bandwidth in the SANMIGUEL and TEMPLE scenes, which we consider the most relevant workloads for an interactive ray tracing architecture. The gains on the CROWN are still significant, though somewhat less due to the large amount of small triangles. Interestingly, our method performs the best on the PLANTS trace, which features several small BVHs using instancing. Finally, HAIRBALL is a pathological case, where the extreme depth-complexity of thin geometry makes our clustering heuristic based on surface area ineffective: on this scene we realize only 13% reduction in node bandwidth. When combined with the triangle bandwidth, our total improvement in the L1 bandwidth is less, but still above 13% for all but one scenes.

L2 bandwidth When analyzing the L2 bandwidth, we can observe that the *miss rate* does not improve. However, this is only because we already filtered out more bandwidth in the L1 cache, thus the remaining accesses to L2 are less coherent. Overall, we can still reduce the L2 to L1 traffic with more than 10% with the TEMPLE, CROWN and PLANTS scenes. Our heuristic fails for HAIRBALL, while in the SANMIGUEL trace the rays are so coherent that they produce similarly good results using all layouts.

The impact of leaf extension nodes Besides node traversal, the triangle intersections also produce significant bandwidth. While we could adopt mesh layout optimization techniques, we can partially address this limitation by storing the vertex indices in our leaf nodes. This alternative layout is motivated by the fact that vertex indices are not shared by different triangles (unlike vertex positions), and since we store the bounding box in the parent node, traversing a primitive leaf node always results in accessing the triangle. Since this additional information does not fit into 8 bytes, we increase the size of the primitive leaves to 16-bytes that are loaded in two 8-byte transactions by the leaf unit.

It is interesting to compare the previous bandwidth results to the measurements using these extension nodes (Fig. 8B). This removes the index fetches from the PU, and moves it to the LU, which loads the indices with a second node fetch from $L1_{Node}$. Note that while indices take only 12 bytes compared to the 36 bytes required by the vertex fetches before the intersections, the L2 to L1 triangle traffic reduces with more than 50% in all scenes! This is because vertices are shared among multiple triangles, so their hit rate is much better in the cache, especially with coherent rays. The L1 node bandwidth increases by a small margin due to the additional queries, but the total bandwidth is still lower with extension nodes. For the first three test traces our layout can further reduce the bandwidth with extension nodes.

Bandwidth as a function of cache size We also examined how bandwidth and utilization change when using different L1 and L2 sizes. In Fig. 6 we first show that compared to ODFL our clustering achieves a similar relative improvement in L1 bandwidth, regardless the capacity of L1.

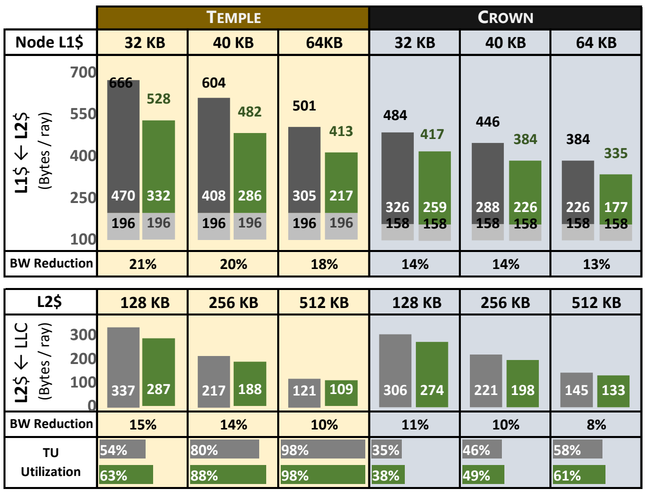


Figure 6: Bandwidth measurements with different L1 and L2 sizes. We compare our method (green) with the standard ODFL (gray).

When scaling the L2 cache with a fixed L1, we see a different trend: as the capacity of L2 increases, the reduction achieved by our method slowly diminishes. Our explanation is that the outstanding misses from L2 become less and less coherent and since more of the frequently traversed nodes reside inside L2, the clustering heuristic cannot predict the outgoing access pattern anymore. There is another interesting trend regarding the utilization of the traversal unit, which increases with the L2 capacity. As we will discuss in Sec. 7.3, this has a critical performance impact. In conclusion, our method greatly improves performance with smaller L2 caches, and has a constant relative improvement in L1 bandwidth-efficiency.

Bandwidth as a function of node size In our design we have opted for an 8-byte node representation, since this has the best alignment for practical cache line sizes. On a 64-byte cache line we can encode 16 boxes (8 node pairs) using 6 bits per plane. This does not mean that our method would not work with larger nodes as well, and we have investigated this problem in our last bandwidth experiment.

There is an interesting tradeoff between node size and bandwidth. Reducing the node sizes trivially shrinks the bandwidth of a given number of traversal steps, however, it also allows less bits in the node quantization, which increases the number of traversal steps (by making the bounds more conservative). In Figure 7 we compare the three layout techniques with different node sizes, also showing the maximum precision they can afford for plane encoding. `ODFL` and `OUR` can use 6 bits on 8 bytes and 8 bits on 10 bytes, respectively, while the second child pointer prevents `CALBVH` from having 8-byte nodes and allows 5 and 7 bit precision on 10 and 12 bytes, respectively. In Fig. 7 we can see that the 8 and 10-byte variants of our method provide similar bandwidth results.

Node Bytes	8		10			12	
Method	ODFL	OUR	ODFL	OUR	CALBVH	CALBVH	
Bits / Plane	6	6	8	8	5	7	
TEMPLE							
T. Nodes	96.9	103.7	87.1	90.5	110.1	91.0	1/ray
L2$ → L1$	441.41	371.24	417.75	355.70	476.89	385.57	B/ray
LLC → L2$	120.46	108.22	115.88	102.22	148.21	113.15	B/ray
CROWN							
T. Nodes	68.0	70.8	63.3	65.5	73.8	64.6	1/ray
L2$ → L1$	400.18	362.88	415.33	372.84	433.77	395.36	B/ray
LLC → L2$	139.46	130.07	147.49	133.15	156.02	148.49	B/ray

Figure 7: While using larger nodes allows less nodes per cache line, it also reduces the traversal steps due to the improved precision. Compared to ODFL, our method also adds a small amount of steps due to glue node traversal.

7.3. Architectural Implications

Lastly, we move our focus to the overall throughput of our system. While the reduction we realized in cache bandwidth can be very important for lowering the energy consumption, our ultimate motivation is to design a system with a high throughput (rays/s). The purpose of this section is to show that bandwidth by itself is not a sufficient indicator of performance, because the system has multiple potential bottlenecks. We show some of the important factors of the hardware that may limit utilization, and how the system scales with adding more traversal clusters.

The overall throughput of the system is based on the utilization of the traversal units. In some cases low utilization can be addressed by just increasing the number of threads (latency hiding), but in other cases the system has an inherent bottleneck. Two examples for this are the ratio of different units and the theoretical throughput of the buses.

Ratio of units: While the number of T/L/P units within a cluster can be chosen conservatively, we deemed it wasteful to scale for worst-case performance and set the ratio of T/P to 8/1: most traces we are aiming to process contained this ratio of triangles to internal nodes. However, this means that we are going to be bottlenecked on other traces where the ratio is less. The examples for this are the CROWN and HAIRBALL traces. As we can observe in Fig. 9, the utilization on these workloads is limited, even with only a few Traversal Clusters. For interactive rendering, we considered these scenes outliers.

Bus utilization: In our final experiment we analyze the scalability of the system, given by the restriction of the data ports. In Fig. 9, we can see how the utilization and ray throughput changes when we increase the traversal clusters in the system. Ideally one can hope for a linear scaling, but unfortunately the frequency of transactions on the L2 to LLC bus (shared by all clusters) asymptotically reaches its limit. In our case this is one 64-byte cache line per cycle. Many of our scenes become L2 bandwidth-bound when we scale the number of clusters to 8, and this is where the reduction of storing indices in leaf nodes (Sec. 7.2) can bring major benefits.

8. Conclusion

We study the impact of BVH compression based on a practical memory hierarchy and show that the quantization of bounding volumes alone does not achieve its full potential for bandwidth reduction. We address this problem by introducing a novel BVH layout that achieves better compression of BVH nodes as well as improved cache line locality and demonstrate a significant reduction in overall bandwidth.

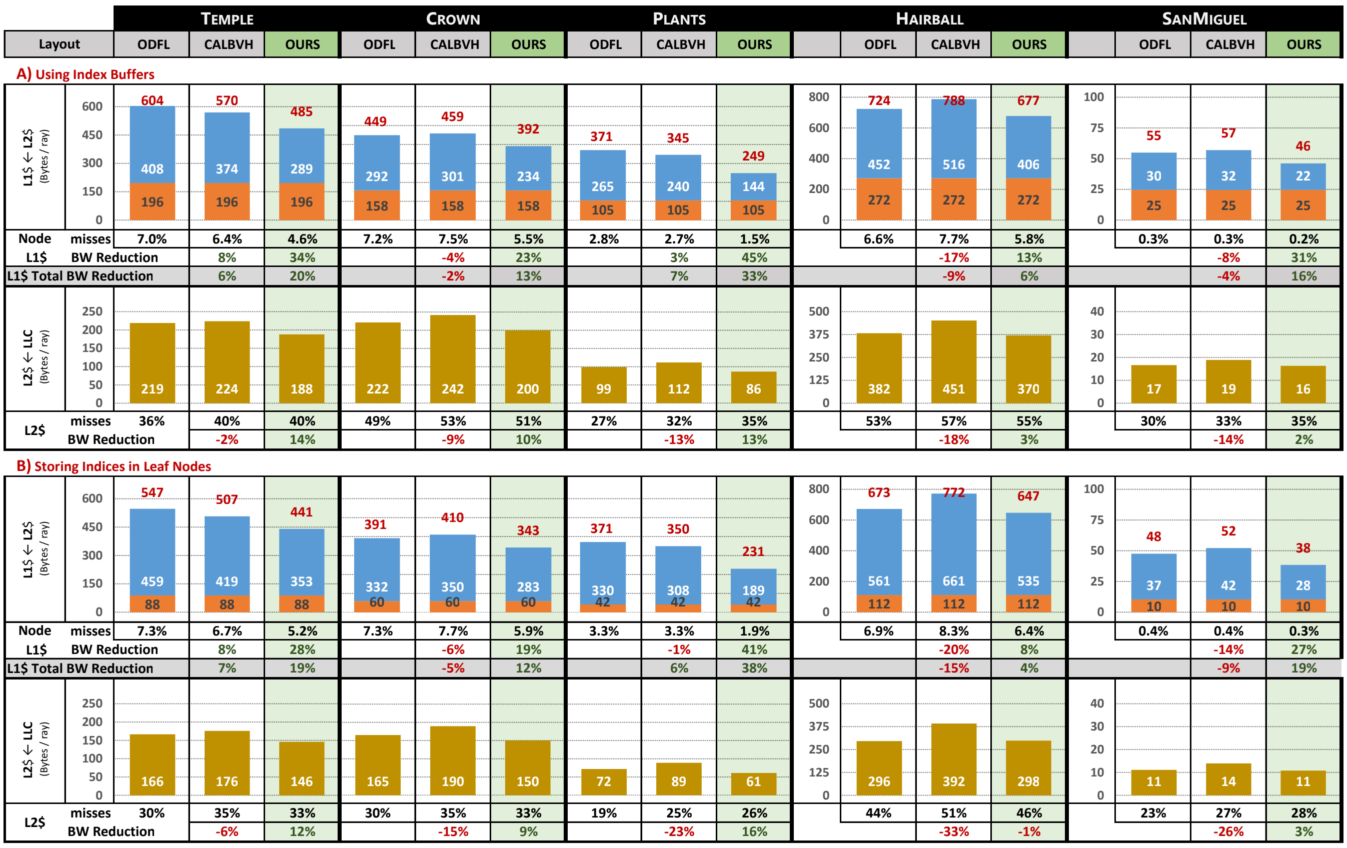
We also discuss important architectural implications of BVH compression. First, the reduced bandwidth throughout the memory hierarchy permits sharing of caches. Second, the significantly reduced size of the BVH nodes compared to the size of a ray makes global ray reordering schemes less appealing. Lastly, we show that a ray throughput of several hundred million rays per second can be achieved with a reasonable bandwidth.

Currently we have not optimized our implementation for BVH build performance, but we believe that an efficient parallel implementation should be possible due to the greedy nature of our algorithm. In the future we would also like to investigate compression techniques for primitive data. We are hopeful that addressing these few remaining challenges can user in an era of real time ray tracing in the near future.

Acknowledgements We would like to thank Charles Lingle, David Blythe, and Tom Piazza for supporting this research. We are grateful for the valuable contributions of Prasoonkumar Surti, David Baldwin, and Marco Salvi. We thank various artists for making their scenes available for the community. The TEMPLE scene is courtesy of Epic Games, Inc. The CROWN scene is courtesy of Martin Lubich, and SANMIGUEL was modeled by Guillermo M. Leal Llaguno.

# clusters		TEMPLE	CROWN	PLANTS	HAIRBALL	SANMIGUEL	
Using **Index buffers**, **40 KB** Node L1$, **32 KB** Tri L1$							
2	TU Util.	99	70	93	51	97	%
	Bus Util.	29	38	8.8	35	2.2	%
	Mrays/s	174.5	186.6	111.1	92.2	105.1	
4	TU Util.	98	61	92	50	97	%
	Bus Util.	59	68	18	70	4.5	%
	Mrays/s	347.8	326.0	218.8	179.3	209.3	
8	TU Util.	79	34	87	32	97	%
	Bus Util.	96	87	35	94	8.9	%
	Mrays/s	555.9	367.4	415.4	228.6	416.9	
Indices in leaf nodes, **48 KB** Node L1$, **24 KB** Tri L1$							
2	TU Util.	95	63	90	44	94	%
	Bus Util.	21	28	6.1	25	1.4	%
	Mrays/s	167.8	167.7	107.5	78.8	101.5	
4	TU Util.	95	59	88	43	94	%
	Bus Util.	43	53	12	49	2.8	%
	Mrays/s	335.0	316.0	209.3	154.9	202.1	
8	TU Util.	91	41	86	38	94	%
	Bus Util.	83	80	24	91	5.6	%
	Mrays/s	642.9	436.8	409.9	273.1	407.0	

Figure 9: The scalability of our system under different workloads. The throughput can grow near-linearly up to the limit of the LLC to L2 bus. Storing indices in leaf nodes reduces pressure on the bus and allows better scaling.



A) Using Index Buffers

Scene	Layout	L1$ ← L2$ total	Node	Triangle	Node misses	L1$ BW Reduction	L1$ Total BW Reduction	L2$ ← LLC	L2$ misses	L2$ BW Reduction
Temple	ODFL	604	408	196	7.0%			219	36%	
Temple	CALBVH	570	374	196	6.4%	8%	6%	224	40%	-2%
Temple	OURS	485	289	196	4.6%	34%	20%	188	40%	14%
Crown	ODFL	449	292	158	7.2%			222	49%	
Crown	CALBVH	459	301	158	7.5%	-4%	-2%	242	53%	-9%
Crown	OURS	392	234	158	5.5%	23%	13%	200	51%	10%
Plants	ODFL	371	265	105	2.8%			99	27%	
Plants	CALBVH	345	240	105	2.7%	3%	7%	112	32%	-13%
Plants	OURS	249	144	105	1.5%	45%	33%	86	35%	13%
Hairball	ODFL	724	452	272	6.6%			382	53%	
Hairball	CALBVH	788	516	272	7.7%	-17%	-9%	451	57%	-18%
Hairball	OURS	677	406	272	5.8%	13%	6%	370	55%	3%
SanMiguel	ODFL	55	30	25	0.3%			17	30%	
SanMiguel	CALBVH	57	32	25	0.3%	-8%	-4%	19	33%	-14%
SanMiguel	OURS	46	22	25	0.2%	31%	16%	16	35%	2%

B) Storing Indices in Leaf Nodes

Scene	Layout	L1$ ← L2$ total	Node	Triangle	Node misses	L1$ BW Reduction	L1$ Total BW Reduction	L2$ ← LLC	L2$ misses	L2$ BW Reduction
Temple	ODFL	547	459	88	7.3%			166	30%	
Temple	CALBVH	507	419	88	6.7%	8%	7%	176	35%	-6%
Temple	OURS	441	353	88	5.2%	28%	19%	146	33%	12%
Crown	ODFL	391	332	60	7.3%			165	30%	
Crown	CALBVH	410	350	60	7.7%	-6%	-5%	190	35%	-15%
Crown	OURS	343	283	60	5.9%	19%	12%	150	33%	9%
Plants	ODFL	371	330	42	3.3%			72	19%	
Plants	CALBVH	350	308	42	3.3%	-1%	6%	89	25%	-23%
Plants	OURS	231	189	42	1.9%	41%	38%	61	26%	16%
Hairball	ODFL	673	561	112	6.9%			296	44%	
Hairball	CALBVH	772	661	112	8.3%	-20%	-15%	392	51%	-33%
Hairball	OURS	647	535	112	6.4%	8%	4%	298	46%	-1%
SanMiguel	ODFL	48	37	10	0.4%			11	23%	
SanMiguel	CALBVH	52	42	10	0.4%	-14%	-9%	14	27%	-26%
SanMiguel	OURS	38	28	10	0.3%	27%	19%	11	28%	3%

Figure 8: General bandwidth measurements on all test scenes. The cache sizes were: Node L1 40 KB, L2 256 KB, Triangle L1 32 KB. On the L1 bus we color-coded the triangle-bandwidth in orange and the node-bandwidth in blue. We can observe a major improvement in bandwidth when the indices are stored in leaf nodes.

References

[AK10] AILA T., KARRAS T.: Architecture Considerations for Tracing Incoherent Rays. In *High-Performance Graphics* (2010), pp. 113–122. 2, 7

[AKL13] AILA T., KARRAS T., LAINE S.: On Quality Metrics of Bounding Volume Hierarchies. In *High-Performance Graphics* (2013), pp. 101–107. 2

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics* (2009), pp. 145–149. 2

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic ray stream traversal. *ACM Trans. Graph. 33*, 4 (July 2014), 151:1–151:9. 2

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *EGSR'08 Proceedings of the Nineteenth Eurographics Conference on Rendering* (2008), pp. 1225–1233. 2

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug 2008), pp. 35–40. 2

[EW11] ERNST M., WOOP S.: Ray Tracing with Shared-Plane Bounding Volume Hierarchies. *Journal of Graphics, GPU, and Game Tools, 15*, 3 (2011), 141–151. 1

[FD09] FABIANOWSKI B., DINGLIANA J.: Compact BVH Storage for Ray Tracing and Photon Mapping. In *Proceedings of Eurographics Ireland Workshop* (2009), pp. 1–8. 1, 2, 5

[fSK14] ÁFRA A. T., SZIRMAY-KALOS L.: Stackless multi-bvh traversal for cpu, mic and gpu ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140. 2

[GI99] GIL J., ITAI A.: How to pack trees. *Journal of Algorithms 32* (1999), 113–127. 2, 3

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug 2008), pp. 59–66. 2

[Hav97] HAVRAN V.: Cache sensitive representation for the bsp tree. In *Compugraphics* (1997), vol. 97, pp. 369–376. 2

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (2007), I3D '07, ACM, pp. 167–174. 2

[Kee14] KEELY S.: Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics* (2014), pp. 29–40. 2, 6, 7

[KSBD10] KOPTA D., SPJUT J., BRUNVAND E., DAVIS A.: Efficient MIMD Architectures for High-Performance Ray Tracing. In *IEEE International Conference on Computer Design* (2010), pp. 9–16. 2

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An Energy and Bandwidth Efficient Ray Tracing Architecture. In *High-Performance Graphics* (2013), ACM, pp. 121–128. 2

[Lai10] LAINE S.: Restart Trail for Stackless BVH Traversal. In *High-Performance Graphics* (2010), pp. 107–111. 2, 6

[LSL*13] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: SGRT: A Mobile GPU Architecture for Real-time Ray Tracing. In *High-Performance Graphics* (2013), ACM, pp. 109–119. 2, 5, 6

[Mah05] Mahovsky J. A.: *Ray Tracing with Reduced-precision Bounding Volume Hierarchies*. PhD thesis, 2005. 1, 2, 5

[MB90] MacDonald J. D., Booth K. S.: Heuristics for Ray Tracing using Space Subdivision. 153–165. 2

[NPK*10] Nah J.-H., Park J.-S., Kim J.-W., Park C., Han T.-D.: Ordered Depth-first Layouts for Ray Tracing. In *ACM SIGGRAPH ASIA Sketches* (2010), pp. 55:1–55:2. 2

[NPP*11] Nah J.-H., Park J.-S., Park C., Kim J.-W., Jung Y.-H., Park W.-C., Han T.-D.: T & I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. *ACM Transactions on Graphics, 30*, 6 (2011), 160:1–160:10. 2

[PH10] Pharr M., Humphreys G.: *Physically Based Rendering: From Theory to Implementation*, 2nd ed. Morgan Kaufmann, 2010. 7

[SKKB09] Spjut J., Kensler A., Kopta D., Brunvand E.: TRaX: A Multicore Hardware Architecture for Real-time Ray Tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28*, 12 (2009), 1802–1815. 2

[SWW*04] Schmittler J., Woop S., Wagner D., Paul W. J., Slusallek P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Graphics Hardware* (2004), pp. 95–106. 2

[Tsa09] Tsakok J. A.: Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 151–158. 2

[VSAM16] Vaidyanathan K., Salvi M., Akenine-Möller T.: Watertight Ray Traversal with Reduced Precision. In *High-Performance Graphics* (2016). 1, 2, 5

[Wal07] Wald I.: On Fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing* (2007), IEEE Computer Society, pp. 33–40. 2

[WBB08] Wald I., Benthin C., Boulos S.: Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug 2008), pp. 49–57. 2

[WBW13] Woop S., Benthin C., Wald I.: Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques, 2*, 1 (2013), 65–82. 6

[WSBW01] Wald I., Slusallek P., Benthin C., Wagner M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–165. 2

[WSS05] Woop S., Schmittler J., Slusallek P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics, 24*, 3 (2005), 434–444. 2

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics, 33*, 4 (2014), 143:1–143:8. 7

[YLPM05] Yoon S.-E., Lindstrom P., Pascucci V., Manocha D.: Cache-Oblivious Mesh Layouts. In *ACM Transactions on Graphics,* (2005), vol. 24, ACM, pp. 886–893. 4

[YM06] Yoon S.-E., Manocha D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum, 25*, 3 (2006), 507–516. 1, 2, 3, 5, 8