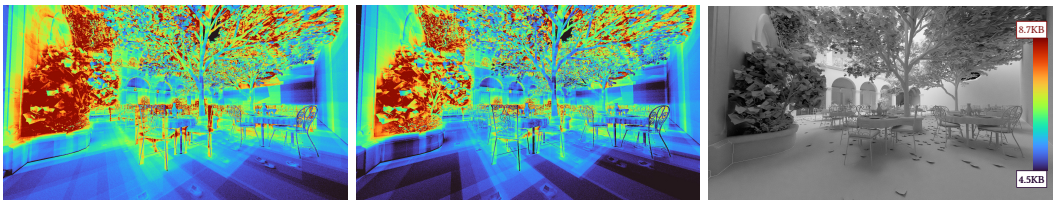
Memory-Efficient Bounding Volume Hierarchies with Merged Nodes for Hardware Ray Tracing

JACOB HAYDEL, University of Utah, USA
ANDREW KENSLER, Advanced Micro Devices, Inc., USA
ERIK BRUNVAND, University of Utah, USA
CEM YUKSEL, University of Utah, USA



(a) Baseline wide BVH
Memory traffic: 6.25 KB/path
BVH size: 588 7.32
Render time: 7.32 ms

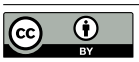
(b) Our BVH with Merged Nodes
Memory traffic: 5.26 KB/path
BVH size: 304 MB
Render time: 5.00 ms

(c) Path traced image
Memory traffic: 15.8% less with ours
BVH size: 48.3% less with ours
Render time: 31.7% faster with ours

Fig. 1. *Heatmap visualization of total memory access per pixel, comparing **(a)** a state-of-the-art wide BVH [Vaidyanathan et al. 2022; Ylitie et al. 2017] optimized for GPU ray tracing to **(b)** our wide BVH with merged nodes. The visualization for the baseline BVH has significantly more red and yellow pixels than ours, particularly around geometric discontinuities, indicating where most memory accesses are generated. Our BVH produces a 15.8% reduction in total memory traffic, a 48.3% reduction in BVH size, and a 31.7% reduction in render time.*

The performance of hardware-accelerated ray tracing on modern GPUs, even with advances in traversal hardware and BVH compression, is fundamentally limited by memory bandwidth. This makes memory traffic—not compute—the primary bottleneck in ray tracing, and demands data structures and construction algorithms that are explicitly optimized for bandwidth efficiency. We introduce a new block-based representation for wide bounding volume hierarchies (BVHs) that directly targets this bottleneck. Our approach organizes multiple primitives and internal nodes into compact, bandwidth-efficient blocks, reducing the number and cost of memory transactions during traversal. Unlike conventional layouts, our representation enables the merging of both internal and leaf nodes, forming composite bounding volumes that amortize memory accesses across larger portions of the hierarchy. To further align BVH construction with hardware realities, we introduce a memory-centric reformulation of the surface area heuristic (SAH). Rather than modeling traversal cost in terms of compute, our formulation estimates the cost of data movement, yielding a metric that more accurately predicts performance on modern GPUs. Under this model, our merged-node representation achieves substantial reductions in memory traffic. We describe both an optimal construction algorithm and an efficient greedy variant that leverage our representation and bandwidth-driven cost model. Across a range of scenes, our approach reduces acceleration structure size and significantly lowers data movement per ray, resulting in consistently faster rendering. These results demonstrate that treating memory bandwidth as a first-class design constraint leads to more efficient ray tracing acceleration structures.

Authors' Contact Information: Jacob Haydel, University of Utah, Salt Lake City, UT, USA, jcbhaydel@gmail.com; Andrew Kensler, Advanced Micro Devices, Inc., Seattle, WA, USA, andrew.kensler@amd.com; Erik Brunvand, University of Utah, Salt Lake City, UT, USA, elb@cs.utah.edu; Cem Yuksel, University of Utah, Salt Lake City, UT, USA, cem@cemyuksel.com.



ACM 2577-6193/2026/7-ART48
https://doi.org/10.1145/3820018

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Ray tracing**; **Graphics processors**.

ACM Reference Format:
Jacob Haydel, Andrew Kensler, Erik Brunvand, and Cem Yuksel. 2026. Memory-Efficient Bounding Volume Hierarchies with Merged Nodes for Hardware Ray Tracing. *Proc. ACM Comput. Graph. Interact. Tech.* 9, 4, Article 48 (July 2026), 18 pages. https://doi.org/10.1145/3820018

1 Introduction

Hardware dedicated to ray tracing is now widely available in modern GPUs. These devices contain an impressive amount of compute resources, including custom traversal units that can automatically traverse acceleration structures and perform intersection testing. Still, ray tracing remains expensive, and only a small number of rays per pixel can be handled within the performance requirements of typical real-time rendering applications.

However, adding more compute resources yields diminishing returns. This is because, for GPUs with sufficient compute power, ray traversal performance is dominated by memory behavior—specifically, cache and DRAM traffic. This problem is further exacerbated by memory contention from shading operations such as attribute interpolation and texturing. Consequently, improving hardware-assisted ray traversal requires optimization strategies that explicitly target memory performance, which has been a challenge.

The core of addressing this challenge is improving the ray tracing acceleration structure used to represent the scene primitives and accelerate ray queries. State-of-the-art methods for hardware ray tracing use a *bounding volume hierarchy* (BVH) with a wide branching factor to construct shallower trees with fewer internal nodes (as compared to binary trees), and reduced-precision representations to minimize the storage of bounding box data [Vaidyanathan et al. 2022; Ylitie et al. 2017]. While such BVHs greatly improve the ray tracing performance, as compared to a full-precision binary BVH, the way they are constructed often produces inefficiencies for hardware traversal. Specifically, the BVH's wide branching factor can often be underutilized, resulting in a considerable fraction of empty nodes in the tree. More importantly—without lossy compression [Barczak et al. 2024]—primitive encoding in the leaf nodes is often highly inefficient, resulting in a large amount of duplicated data to avoid costly indirections. Furthermore, the *surface area heuristic* (SAH) that virtually all BVH builders aim to optimize is designed for compute performance, as opposed to data movement, which is more critical for hardware ray tracing performance.

In this paper, we introduce a novel wide BVH representation with block-based storage that utilizes *merged nodes* to improve the memory efficiency of the acceleration structure. Our merged nodes combine multiple nodes of the BVH, effectively forming bounding volumes that are represented as the union of multiple boxes. They provide both storage efficiency and a reduction in data movement when a ray intersects multiple merged nodes. Our structure is particularly effective at merging leaf nodes into *primitive blocks* spanning multiple triangles, thereby greatly reducing data duplication. This reduces both storage cost and data movement during ray traversal. In addition, we propose a modification to the SAH formulation that is based on data movement cost rather than computational cost. Our *memory-based SAH* (MSAH) offers a better correlation with real-world performance than SAH costs based on computation. Furthermore, our merged nodes maintain the MSAH cost, thereby improving memory efficiency without reducing BVH quality.

An acceleration structure is only practical if efficient build methods exist for constructing high-quality trees (in terms of rendering performance). Therefore, we also describe a dynamic programming method that can produce memory-friendly wide BVHs with our primitive blocks. Additionally, we introduce a fast greedy builder that takes advantage of our merged blocks to recover performance lost due to its greedy decisions, resulting in similar quality BVHs in a fraction of the build time.

Because vendors of commercial GPUs do not expose their internal BVH formats, nor allow the use of user-described BVH formats, we evaluate our contributions using Arches [Haydel et al. 2025], a cycle-level hardware simulator. This allows us to render a variety of scenes with different workloads, ranging from primary rays with strong coherence to tertiary rays with strong divergence, on simulated hardware that closely tracks contemporary GPUs but with our proposed data structures. Our results show that our acceleration structure significantly reduces the data movement (Fig. 1), which leads to substantial (up to 5× in some cases) reductions in ray traversal time. In addition to these immediate savings in ray tracing cost, our contributions have the potential to offer further improvements in rendering performance: the bandwidth saved on ray tracing can then be allocated to other demands, such as shading, rasterization, and compute.

2 Background

There is a large body of work on BVHs and ray tracing acceleration structures more broadly [Meister et al. 2021]. In this section, we briefly summarize the most closely related prior work on BVHs, starting with the surface area heuristic formulation we will modify.

2.1 Surface Area Heuristic (SAH)

SAH is one of the core concepts behind building ray tracing acceleration structures, as it provides a method for estimating the cost of a given tree. The SAH works by estimating the conditional probability of visiting a node via its surface area. The expected cost of a given tree can then be expressed as the cost of visiting each node multiplied by the likelihood of visiting the respective node. More specifically, the SAH cost of an entire tree can be expressed as

$$C_{\text{SAH}} = \sum_{n \in T} A_n c_n \,, \tag{1}$$

where A_n is the surface area of the node divided by the surface area of the root node, and c_n is the cost of intersecting the node. Here, c_n is expressed as

$$c_n = \begin{cases} c_{\text{node}}, & \text{if internal} \\ P_n c_{\text{prim}}, & \text{if leaf} \end{cases} , \tag{2}$$

where c_{node} is the computational cost of traversing the node data, which is the cost of ray-box intersection for a bounding volume, c_{prim} is the cost of ray intersection with a primitive, and P_n is the number of primitives contained in the leaf node.

Minimizing this heuristic tends to produce trees that provide ray tracing efficiency for a wide variety of ray distributions. However, doing this well and in a time-efficient manner is nontrivial and, therefore, has been the topic of much of the later research on acceleration structure building.

In our work, we propose a modification to the traditional formulation of the SAH, which is better suited to hardware-accelerated ray traversal.

2.2 Prior Work

BVHs with wide branching factors are commonly used today, not only for software ray tracing, but also for hardware-accelerated implementations. Such wide BVHs are typically built starting with a binary BVH.

High-quality BVHs. A common method for building SAH-optimized binary BVH is a top-down divisive build. Such builders work by recursively splitting primitives into two groups based on a given spatial plane. A common technique is to simply check every unique axis-aligned splitting plane and select the one with the lowest SAH cost [MacDonald and Booth 1990]. This produces a high-quality tree, but it is a greedy algorithm that does not necessarily produce an SAH-optimal

tree. Tree rotations [Kensler 2008] were introduced as a method for iteratively improving the SAH cost of the tree post-construction. Trees optimized with tree rotations can produce higher-quality BVHs than traditional top-down build methods alone. Similarly, reinsertions [Bittner et al. 2013] were proposed as another method for optimizing trees by iteratively reducing SAH. The Split BVH (SBVH) [Stich et al. 2009] structure was proposed to further improve the tree's quality by introducing spatial splits similar to those found in k-d trees. This can reduce SAH at the cost of duplicating primitive data and, therefore, increasing tree size.

Fast BVH builds. Rapid construction of BVHs on the GPU is a crucial problem for real-time ray tracing applications, particularly in dynamic scenes. One quick solution for dynamic scenes is refitting a previously constructed BVH by preserving the tree structure and only adjusting the bounding box data [Wald et al. 2009]. Yet, with large animations, refitting can significantly degrade the quality of the BVH. Therefore, such applications are often forced to rebuild the scene BVHs at run time to support animated geometry. One of the fastest methods for BVH construction on the GPU is Linear BVH (LBVH) [Apetrei 2014; Karras 2012; Lauterbach et al. 2009]. LBVH works by arranging primitives along a space-filling curve. A BVH can then group primitives by common prefix. The resulting trees are fast to construct and of reasonable quality. Several other methods have focused on refining BVHs on the GPU. Parallel reinsertion BVH [Meister and Bittner 2018] is a GPU implementation of reinsertion that can be used to produce high-quality BVH on the GPU at considerable cost. Agglomerative Treelet Restructuring [Domingues and Pedrini 2015], Parallel Local Order Clustering [Benthin et al. 2022], and Hierarchical Parallel Local Order Clustering (H-PLOC) [Benthin et al. 2024] are all methods that use agglomerative clustering to produce higher-quality BVHs relatively quickly. Of these, H-PLOC is the fastest, only introducing minimal overhead compared to LBVH while producing higher-quality trees.

Wide BVHs. Since wide BVHs improve ray tracing performance over binary BVH, binary-to-wide BVH conversion has been the focus of some prior work. Early work on wide BVH conversion targeted SIMD traversal on the CPU [Dammertz et al. 2008; Ernst and Greiner 2008; Wald et al. 2008]. The idea behind this work was to produce BVHs with higher arity such that the intersection code could fully leverage SIMD hardware. Notably, Wald et al. [2008] also propose a modified version of the SAH cost that does not grow linearly with primitive count. The primary objective of this change is to reflect the computational cost of the intersection using SIMD instructions, as opposed to our modification that is based on data movement instead of compute cost. Later work focused on the construction of high-quality compressed wide BVHs for software traversal on the GPU. Ylitie et al. [2017] describe a specific compressed BVH that is well-suited to software traversal, alongside a dynamic programming method to build these BVHs. However, they do not address primitive encoding, which is a crucial component that can significantly impact the resulting ray tracing performance. Saed et al. [2022] describe a modified version of this wide BVH structure as a proxy for the proprietary BVH format used for hardware ray tracing on NVIDIA GPUs. More recently, Benthin et al. [2024] and Barbier and Paulin [2025] proposed GPU implementations of top-down and bottom-up greedy collapses, respectively.

Lossy geometry in BVHs. Recently, *dense geometry format* (DGF) [Barczak et al. 2024] was introduced as a lossy cluster compression for primitive encoding targeting hardware ray tracing. This format focuses on quantizing vertex data on a global grid and then compressing this data into 128-byte blocks using triangle strips. The result is highly compressed lossy primitive data, which can be used to represent leaf node data in a BVH. Our primitive representation, in comparison, is lossless, though DGF leaves can be incorporated into our representation if lossy compression is permitted.

3 Memory-Efficient BVH with Merged Nodes

Wide BVHs are typically formed by collapsing the nodes of a binary BVH. Therefore, wide BVHs form shallower trees with fewer nodes, as compared to their binary counterparts. This reduces data movement during rendering, at the cost of additional bounding box tests compared to the corresponding binary BVH. Yet, this additional computational cost can be absorbed using vector instructions with software ray tracing or dedicated traversal and intersection hardware on GPUs.

The improvements a wide BVH can offer are correlated with its *node fullness*, the percentage of non-empty child nodes in each wide node. The dynamic programming approach [Ylitie et al. 2017] is slow, but it can produce high-quality BVHs with relatively high node fullness. Faster greedy builds [Barbier and Paulin 2025; Benthin et al. 2024], on the other hand, achieve lower node fullness. More importantly, wide BVHs form significantly more leaf nodes than internal nodes. Therefore, the efficiency of the primitive format plays an important role in both storage and performance.

Our wide BVH representation with merged blocks addresses node fullness and primitive encoding, with a strong focus on a node and primitive block structure that matches the characteristics of the memory system. This reflects the memory-bound nature of hardware approaches to BVH traversal, where the primary performance enhancement comes from more efficient memory access.

First, we describe our *primitive blocks* that efficiently store multiple primitives without relying on compression (Sec. 3.1). Then, we introduce our *node blocks* with *merging* that allow us to improve the fullness of both node blocks and primitive blocks without negatively impacting the quality of the tree (Sec. 3.2). This is accomplished by forming node blocks that are bounded by the union of multiple bounding boxes, instead of a single bounding box, as in traditional BVH formats. Our representation is designed to reduce data movement during rendering and allows for evaluating the quality of the tree using a new memory-based SAH formulation that better correlates with actual hardware rendering performance than the traditional SAH that is based on the computational cost (Sec. 3.3). We then describe improvements to wide BVH construction from a binary BVH by incorporating the BVH representation into the construction decision, along with our memory-based SAH formulation. The results are an optimal builder based on dynamic programming and a faster alternative that can achieve a similar BVH quality using a series of greedy algorithms (Sec. 3.4). Finally, we explain how the merging operation is performed after a wide BVH is constructed while preserving tree quality (Sec. 3.5).

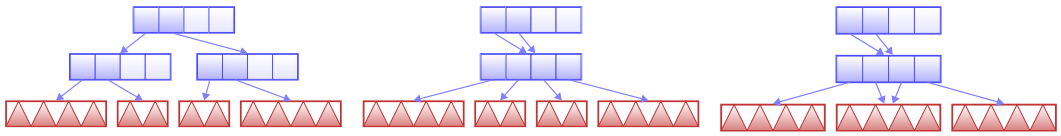
3.1 Primitive Blocks

The typical triangle mesh storage, using a vertex buffer and an index buffer, forms a compact data structure. When the primitive data in a BVH is stored in this format, however, it results in frequent indirect memory accesses that hurt hardware ray tracing performance. Therefore, a common solution is to store triangle pairs as four vertex positions [Vaidyanathan et al. 2022], thereby eliminating the overhead of indirection. However, this comes at a cost of duplicating the vertex data many times over (typically about 4×).

Our *primitive blocks* offer a solution: we use the same memory-efficient format, consisting of a vertex buffer and index buffer, but only within small memory blocks of a fixed size (i.e., locally). The triangles in the primitive block are represented similarly to meshlets using an *index buffer* with local indices to the vertex positions stored in a *vertex buffer*. This eliminates the vertex duplication within the primitive block, thereby reducing (though not eliminating) the global vertex duplication in the BVH. In addition, each primitive block also stores a global primitive ID. All primitives in the block are assigned consecutive global IDs, avoiding the need for storing explicit primitive IDs for each triangle.

(a) Before node merging **(b)** After node merging

Fig. 2. *Leaf merging.* ***(a)*** *The first two leaf nodes are merged into* ***(b)*** *a single primitive block. As a ray hitting either of the original bounding boxes stored in the parent node produces a single traversal step to the merged primitive block, spatially the bounding volume of the merged primitive block is effectively their union.*



(a) Before node merging **(b)** After node merging **(c)** After an additional leaf merging

Fig. 3. *Node merging.* ***(a)*** *Two internal nodes are merged into* ***(b)*** *a single node block.* ***(c)*** *Node merging produces a new opportunity for leaf merging lower in the tree.*

3.2 Node Blocks with Merged Nodes

In our representation, a node block contains a collection of internal and leaf nodes. Internal nodes point to other node blocks, and leaf nodes point to primitive blocks. We use compressed bounding boxes to pack as many nodes as possible into a fixed-size node block.

Most importantly, our node blocks allow *node merging* to improve the fullness of both node blocks and primitive blocks. When two leaf nodes in a node block are merged, a single primitive block containing their combined triangle list is generated (Fig. 2). Similarly, when internal nodes are merged, they all point to the same node block that contains their collective child nodes (Fig. 3). Crucially, in either case, the merging operation preserves the bounding boxes of the original nodes. This distinguishes merging from the *collapsing* used for wide BVH construction, which combines multiple nodes into a single node with a single bounding box. In the case of merging, we end up with a shared node or primitive block bounded by the union of the original bounding boxes.

The merged nodes are easy to identify during ray traversal, because all nodes of a merged set are contained within the same node block. Thus, duplicate traversal can be trivially avoided during stack push by merging stack entries that point to the same block. This ensures that merged blocks are not visited multiple times during traversal.

This node merging serves two purposes:

(1) Internal nodes that are not full can be packed together with other partially-filled internal nodes. This reduces the wasted space due to partially-filled nodes and increases the block fullness of the tree. Yet, this is achieved while maintaining the bounding boxes of the merged nodes.

(2) Primitive blocks and internal nodes can be bounded as a union of multiple boxes, as opposed to a single box that contains the merged subtree. This has the potential to form a tighter fit than a single box.

Because merged interior and leaf nodes maintain their bounding boxes, merging does not degrade the tree's quality. This is in contrast to collapsing, which may produce looser bounds.

3.3 Memory-Based Surface Area Heuristic (MSAH)

Moving from software to hardware traversal has an important implication for the SAH cost. Since intersection hardware usually runs faster than the memory system, hardware ray traversal is generally memory-bound. Thus, traversal cost is more accurately expressed in terms of the amount of memory traffic generated rather than the computational cost.

Our memory-based SAH (MSAH) replaces the cost of a node in Eq. 2 with one that directly expresses the cost of accessing the node data as

$$c_n = \begin{cases} s_{\text{node}}, & \text{if internal} \\ B_n\, s_{\text{prim}}, & \text{if leaf} \end{cases}, \tag{3}$$

where s_{node} is the size of a node block, s_{prim} is the size of a primitive block, and B_n is the number of primitive blocks needed to encode the leaf node. Since our BVH can only have a single primitive block per leaf node, we always have $B_n = 1$. Importantly, in the case of our BVH, this means that the cost of visiting a primitive block is not directly proportional to the number of triangles it contains.

A key property of our node merging is that it does not negatively impact MSAH. When two nodes are merged, their bounding boxes are preserved within the parent node block, so their intersection probabilities can still be estimated using the same surface areas of the same boxes. The cost of accessing a node also remains fixed, using the same node block size or primitive block size, regardless of whether the same node block or primitive block is shared by another node. As a result, using MSAH while building wide BVHs never prevents merging, leading to improved rendering performance, as compared to the traditional SAH.

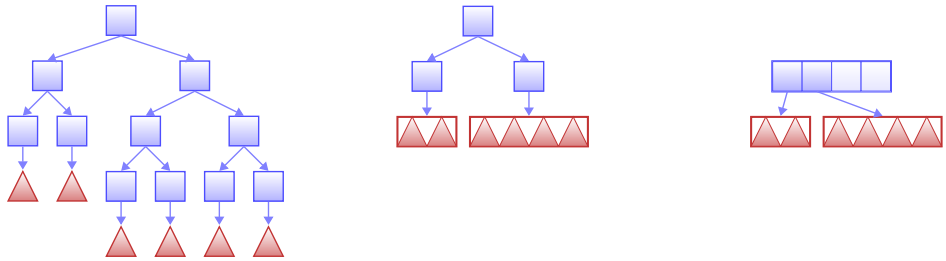
Using our MSAH for constructing an acceleration structure requires knowledge of how the internal and leaf nodes are stored. If the storage format is ignored by assuming that the storage cost scales linearly with the number of primitives, our MSAH cost becomes practically identical to SAH, except for the relative costs of $c_{\text{node}}/c_{\text{prim}}$ and $s_{\text{node}}/s_{\text{prim}}$, eliminating most of the benefits.

3.4 Memory-Aware Wide BVH Construction

Wide BVHs are typically constructed from a binary BVH by *collapsing* its nodes. While building the binary BVH, SAH and MSAH yield identical splitting decisions. The only difference is in the termination condition for the leaf nodes, which is often deferred to wide BVH conversion.

A high-quality wide BVH can be formed from a binary BVH by employing dynamic programming [Ylitie et al. 2017]. The algorithm starts by computing a cost tree in a bottom-up manner using the SAH, and then performs a top-down pass to realize the best tree.

To produce MSAH-optimal trees, we employ a similar approach. To build our cost tree, we start by traversing the binary BVH in a bottom-up fashion. At each node, we first check whether all triangles in the sub-tree can be encoded in a single primitive block. If they can, then the cost of forming a leaf is computed using our MSAH, reflecting the amount of memory traffic produced by visiting a primitive block. Otherwise, the leaf cost is infinite, as a leaf node that fits in a single primitive block cannot be formed. We then compute the cost of each possible distribution of children from the left and right sub-trees. The cost of forming an interior node is the minimum over all child



(a) Input binary BVH **(b)** BVH after leaf collapse **(c)** Wide BVH after node collapse

Fig. 4. *Construction process using greedy collapses.* ***(a)*** *First, leaves are* ***(b)*** *collapsed upwards to form full primitive blocks,* ***(c)*** *then the tree is collapsed downwards to produce node blocks.*

distributions. Then, like the original algorithm, a second pass is performed to realize the best cost tree in a top-down fashion.

This build method achieves the highest possible quality for binary-to-wide BVH conversion with collapsing. However, it is also computationally expensive, as it requires evaluating whether a primitive block can be encoded for every possible sub-tree that contains fewer than the maximum triangles a primitive block can encode. It also requires $N - 1$ MSAH calculations per node in the binary tree, where N is the branching factor of the target BVH. Furthermore, the resulting cache that holds these calculations can be even larger than the binary BVH itself. Therefore, this method is better suited to offline builds where quality is more important than build performance. The resulting trees provide a lower bound on the MSAH cost for the wide BVH conversion.

In contrast, it is sometimes desirable to rapidly construct BVHs for real-time applications with animated geometry. To that end, we propose a compression-aware greedy method for binary to wide BVH conversion, as illustrated in Fig. 4. This is a two-phase algorithm that begins with a binary BVH containing a single primitive per leaf. We start by traversing the tree in a bottom-up fashion, combining leaf nodes to form larger leaf nodes. We evaluate each node to determine whether the primitives contained in its children can be encoded in a single block. If so, we form a new leaf node and continue up the tree; otherwise, we terminate.

Once all pairs of leaves are combined, the second pass can begin. In the second phase, we perform the top-down BVH collapse proposed by Wald et al. [2008]. For each internal node, we replace the child with the largest surface area with its children until the node has the maximum number of children or all children are leaves.

This two-phase greedy method has several benefits. The first is that the primitive block packing can be integrated into bottom-up builders, such as H-PLOC [Benthin et al. 2024] and LBVH [Apetrei 2014]. This would amount to evaluating whether a primitive block can be formed any time an interior node that parents two leaf nodes is formed. The second is that primitive block encoding can be done directly during this phase, eliminating the need for a second pass like the dynamic method. Furthermore, the resulting binary BVHs are significantly shallower and, therefore, smaller, due to the large leaf nodes formed during the bottom-up phase. This reduces the cost of binary-to-wide BVH conversion. The trees that result from this method are both higher-quality and faster to build than methods that defer the block generation until later in the build.

One drawback of the greedy method is that it suffers from the low node fullness characteristic of top-down collapses. Fortunately, this can be addressed using our lightweight node-merging pass, as described below.

3.5 Node Merging

Node merging is performed after a wide BVH is constructed. At this phase, each node in a node block points to a unique node or primitive block. To merge two interior nodes within a block, we must first evaluate whether both child node blocks can be encoded within a single block. For internal nodes, this is very inexpensive as it only requires knowledge about the total number of nodes in two candidate blocks.

To perform merging (see Fig. 3), we attempt to merge the blocks pointed to by consecutive nodes in a single block, forming a new block only when the current one is full. This greedy method results in suboptimal packing, but makes the merging pass very inexpensive. During node merging, there is no need to evaluate the MSAH cost, because merging preserves the original bounding boxes, leaving MSAH unchanged. Merging should be performed in a top-down fashion since internal node merging produces fuller nodes at lower levels. This means that top-down merging progressively improves the tree's block fullness, creating additional merging opportunities deeper in the tree.

Leaf node merging (see Fig. 2) operates similarly to internal node merging, with one caveat: to merge primitive blocks, the unique vertex set across both primitive blocks must be determined before merging can be performed. This makes leaf node merging more expensive as it involves comparing the vertex data in different primitive blocks to identify shared vertices.

While all build methods can benefit from node merging, top-down greedy collapses are particularly well-suited to benefit from this operation. This is because top-down collapses produce perfectly full nodes higher up in the tree, but very empty nodes in the last level. Since wide BVH nodes mean that the vast majority of nodes reside at the lowest level of the tree, this significantly reduces average node fullness, increasing the BVH footprint. When applied to trees built with our greedy method, node merging is able to recover a large portion of the wasted space in last-level nodes. Furthermore, since all of the unfilled nodes are in the last level, there are no inter-node dependencies, so all last-level nodes can be merged in parallel without traversal.

4 Implementation Details

Our wide BVH format is intended to produce efficient memory access during traversal, which requires careful consideration of data movement and storage granularity in the entire memory hierarchy. Our target architecture, like many modern GPUs, employs a sector cache, which utilizes 128B cache blocks that can be filled at a 32B granularity. This yields several plausible options for primitive and node block size. Specifically, 32B, 64B, and 128B blocks all map well to hardware.

These block sizes represent different tradeoffs: larger blocks have the advantage of increasing opportunities for shared data, while smaller blocks produce a finer access granularity, potentially leading to less memory traffic. For our implementation, we focus on 64B and 128B block sizes, as these represent the most compelling tradeoffs.

4.1 Primitive Block Format

Our primitive blocks must encode three things: vertex data, local index data, and global primitive IDs. Of these, the vertex data tends to be the largest, as full-precision vertices require 12B per vertex. In contrast, the local index buffer requires only a few bits per triangle to index the small vertex buffer, and global IDs can be stored implicitly from a base ID by reordering the attribute arrays such that consecutive primitives in a block have consecutive global IDs.

In practice, this means that our 64B format resembles the primitive encoding proposed by Vaidyanathan et al. [2022], where each block encodes a pair of triangles and a primitive ID.

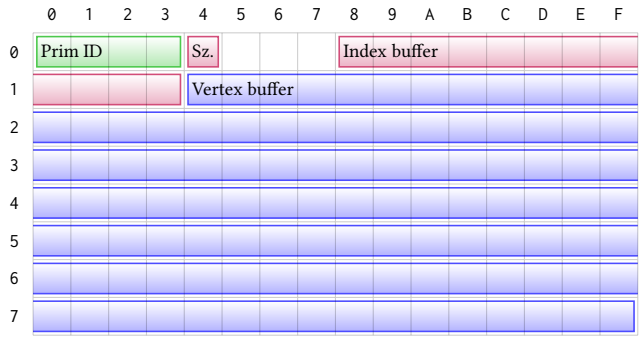


Fig. 5. *An example data layout for our 128B primitive blocks. They contain the first primitive ID, buffer sizes, a local index buffer, and a vertex buffer that stores all vertex positions.*

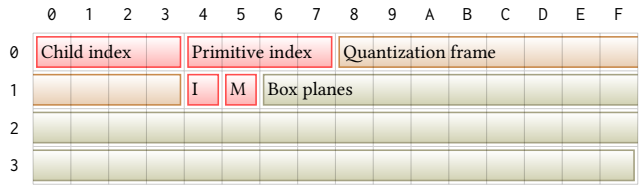


Fig. 6. *An example data layout of our 64B node blocks. Each node contains a base child index, a base primitive index, a quantization frame, a mask to store whether each node is interior or leaf, a mask to store which nodes are merged, and the box planes of 7 quantized boxes.*

Our 128B primitive blocks, in contrast, can encode up to 9 vertices. To fully utilize the vertex data in 128B blocks, we also encode up to 8 triangles using 12 bits per triangle, supporting arbitrary topology. An example 128B primitive block layout is shown in Fig. 5.

It is worth noting that Fig. 5 represents one way of encoding primitive blocks. Other implementations may prefer to encode topology using triangle strips at the cost of some overhead during primitive block construction. Likewise, in cases where reordering attribute arrays is undesirable, primitive indices may be encoded explicitly or via delta encoding. The format in Fig. 5 is intended as a representative example of block-based primitive encoding.

4.2 Node Block Format

Our 64B node block format implementation is illustrated in Fig. 6. This node block uses a quantization frame for compressing the bounding box data, similar to prior work [Ylitie et al. 2017]. In our representation, for each of the three spatial axes, the quantization frame stores a 24-bit floating-point anchor (15-bit significand, 8-bit exponent, and a sign bit) along with an 8-bit exponent for the maximum extent of the quantization frame. Our tests show that 24-bit anchors introduce roughly equivalent quantization error compared to 32-bit anchors, but allow one additional bounding box to fit in a 64B node block. The bounding boxes of the nodes are stored using 8 bits for each of the 6 bounding box planes, and a bit per node is reserved to encode the node type: *internal* or *leaf*.

Each one of our node blocks also stores an *internal node base index* and a *leaf node base index*. The internal and leaf nodes use consecutive indices, starting with the corresponding base indices. Internal nodes use their base index to access child node blocks, while leaf nodes use theirs to access primitive blocks. This scheme of storing two different base indices makes it simple to have node blocks and primitive blocks of different sizes.

(a) Crytek Sponza 0.26M triangles | **(b)** Intel Sponza 3.75M triangles | **(c)** Bistro 2.83M triangles | **(d)** San Miguel 9.97M triangles | **(e)** Hairball 2.88M triangles

Fig. 7. *Viewpoints for the scenes we use in our evaluations.*

A single bit per node is sufficient to encode whether the node references the next block index or shares the index of its merged neighbor. This encoding makes it very easy to guarantee that, during traversal, each block is visited at most once, as all shared pointers are contained in one block. Duplicate traversal is avoided during stack push by retaining only the nearest entry pointing to a given block.

Similar to primitive blocks, node blocks can also be packed into different sizes. Node blocks of 64B have sufficient space for 7 bounding boxes, so they can represent wide BVHs with a branching factor of 7. Going up to 128B allows packing up to 17 boxes, though in our evaluation, we use a branching factor of 16.

5 Evaluation and Results

For our evaluation of traversal performance, we configure Arches [Haydel et al. 2025], a cycle-level simulator, to model a system that is characteristic of a modern GPU. The simulator includes dedicated traversal cores and a detailed model of a cache and DRAM system characteristic of a modern GPU. We specifically use a configuration of the memory system that resembles that of an NVIDIA RTX 2080. While our primary contributions concern the BVH structure and construction, we also modify the simulated traversal cores to handle the merged BVH traversal by updating the simulated cores to contain two node intersection pipelines and two triangle intersection pipelines. This allows them to drive enough memory traffic to saturate the L1 bandwidth. Configured in this way, Arches provides reliable insight into how the theoretical advantages of our BVH, such as reduced memory traffic and a smaller memory footprint, would translate into hardware traversal performance.

For each test, we pre-generate rays from four distributions: primary visibility, mirror reflections, diffuse secondary rays, and diffuse tertiary rays. These distributions represent a spectrum from coherent to very incoherent rays. We measure traversal performance in millions of rays traversed per second (MRays/sec) on five scenes (see Fig. 7), using these ray distributions, and all combinations of the build techniques, and then the overall time to render a frame using our most efficient BVH build techniques.

For our baseline, we use the BVH proposed by Vaidyanathan et al. [2022], which is a 6-way compressed BVH that uses triangle pairs to represent leaf data. This BVH is specifically designed for hardware ray tracing and packs nodes and primitive data into 64B blocks, making it cache-friendly. It represents a state-of-the-art BVH for hardware traversal.

Our BVHs in these tests use 64B compressed node blocks, each containing up to 7 child nodes, and 128B primitive blocks, unless otherwise specified. We compare three construction methods:

- **Dynamic-SAH** uses the build method described by Ylitie et al. [2017], which is a dynamic programming collapse with the traditional SAH formulation.

- **Dynamic-MSAH** uses our dynamic BVH collapse method described in Sec. 3.4, which integrates MSAH into the collapse.
- **Greedy** uses the greedy method described in Sec. 3.4, which tries to produce an MSAH-friendly tree with minimal overhead at build time.

In addition, where applicable, we combine the dynamic build method with a full top-down **Merging** pass and our greedy build with a much cheaper **Node-Merging-Only** pass applied to the last level of the tree.

5.1 BVH Sizes

Examining the BVH footprint and MSAH (Table 1) for the tested trees highlights a shortcoming in prior work: these methods do not produce leaf nodes large enough to fill primitive blocks (as can be seen in the Prims/block column). For the baseline method, we observe good node fullness (i.e., Nodes/block is close to its maximum of 6), but primitive blocks only contain 1.25 to 1.33 triangles on average. The baseline alleviates this slightly by using small leaf nodes, which are only capable of storing up to two triangles, but this results in vertex duplication and thus larger trees that are less cache-friendly.

The Dynamic-SAH version introduces our primitive format without the proposed build methods, but it exacerbates the issue. Dynamic-SAH builds very small leaf nodes, which do not fill the primitive blocks. This nearly doubles the overall BVH footprint, as compared to the baseline.

Introducing merging using the Dynamic-SAH + Merging method dramatically improves the primitive block fullness, from 1.28–1.58 to 4.21–5.42 primitives per block.

Both the Dynamic-MSAH and Greedy methods expand on this, while also decreasing MSAH cost significantly, as compared to Dynamic-SAH. Furthermore, since these methods create larger leaf nodes naturally, they substantially decrease node and primitive footprint (see Node and Leaf Size columns). The merging pass provides additional improvements in node fullness (see Nodes/block and Prims/block), but these improvements are relatively minor, since the MSAH optimization already produces fuller nodes. The Greedy method, in particular, creates such full primitive blocks that there is little need to perform a primitive merging step, which is why we suggest combining it with a Node-Merging-Only step. This comes at only a slight increase in MSAH and memory footprint, as compared to including a primitive merging operation as well (see Dynamic-MSAH + Merging).

5.2 Traversal Performance

Traversal performance tracks with these reductions in MSAH and BVH footprint, as can be seen in Table 2. Several of these results are also visualized in Fig. 8.

Our results show that switching to our BVH format, without modifying the build (i.e., Dynamic-SAH), results in a decrease in performance due to low primitive block fullness, as compared to Baseline. Introducing merging alone without changing the build method (i.e, Dynamic-SAH + Merging) results in a substantial increase in performance, which comes primarily from the decrease in memory traffic *and* memory footprint, validating our focus on the memory behavior as a primary lever for increasing performance.

Introducing our Dynamic-MSAH collapse further decreases memory traffic by producing a tree with a lower MSAH cost (see Table 1). At the same time, these trees are also shallower, which slightly reduces the memory footprint as well (34.0 B/tri to 26.7 B/tri for Intel Sponza). As expected, this leads to an increase in traversal performance. Dynamic-MSAH + Merging provides additional performance improvements in most cases.

Table 1. ***BVH sizes*** *in bytes per triangle and MSAH cost in bytes per ray.*

		Nodes /block	Prims /block	MSAH Cost bytes/ray	Node Size bytes/tri	Leaf Size bytes/tri	Total Size bytes/tri
Crytek Sponza	Baseline	5.54	1.33	1995	10.6	48.3	58.9
	Dynamic-SAH	6.33	1.32	1849	9.1	97.3	106.4
	Dynamic-SAH + Merging	6.35	4.46	1849	9.1	28.7	37.8
	Dynamic-MSAH	5.89	3.91	1322	3.3	32.7	36.1
	Dynamic-MSAH + Merging	5.94	4.64	1322	3.3	27.6	30.9
	Greedy	4.04	4.53	1442	4.6	28.3	32.9
	Greedy + Node-Merging-Only	5.17	4.53	1442	3.6	28.3	31.9
Intel Sponza	Baseline	5.61	1.33	1986	10.5	48.3	58.8
	Dynamic-SAH	6.40	1.30	2418	9.1	98.4	107.5
	Dynamic-SAH + Merging	6.42	5.13	2418	9.1	24.9	34.0
	Dynamic-MSAH	5.93	4.53	1932	2.9	28.2	31.1
	Dynamic-MSAH + Merging	5.99	5.37	1932	2.8	23.8	26.7
	Greedy	3.96	5.29	2032	4.1	24.2	28.3
	Greedy + Node-Merging-Only	5.25	5.29	2032	3.1	24.2	27.3
Bistro	Baseline	5.43	1.26	2716	11.5	50.9	62.3
	Dynamic-SAH	6.26	1.29	2444	9.5	99.6	109.1
	Dynamic-SAH + Merging	6.29	4.21	2444	9.4	30.4	39.9
	Dynamic-MSAH	5.74	3.68	1504	3.7	34.8	38.4
	Dynamic-MSAH + Merging	5.83	4.33	1504	3.6	29.5	33.1
	Greedy	4.14	4.17	1597	4.9	30.7	35.6
	Greedy + Node-Merging-Only	5.16	4.17	1597	3.9	30.7	34.6
San Miguel	Baseline	5.49	1.27	1753	11.3	50.6	61.8
	Dynamic-SAH	6.33	1.28	1610	9.4	99.6	109.0
	Dynamic-SAH + Merging	6.34	4.55	1610	9.3	28.2	37.5
	Dynamic-MSAH	5.63	4.17	1230	3.3	30.7	34.0
	Dynamic-MSAH + Merging	5.74	4.70	1230	3.3	27.2	30.5
	Greedy	4.09	4.57	1299	4.5	28.0	32.6
	Greedy + Node-Merging-Only	5.20	4.57	1299	3.6	28.0	31.6
Hairball	Baseline	5.62	1.25	14914	11.1	51.0	62.1
	Dynamic-SAH	6.84	1.58	11861	7.0	81.2	88.2
	Dynamic-SAH + Merging	6.84	5.42	11861	7.0	23.6	30.6
	Dynamic-MSAH	5.19	5.28	6975	2.9	24.3	27.2
	Dynamic-MSAH + Merging	5.46	5.40	6975	2.8	23.7	26.4
	Greedy	3.97	5.32	7255	4.1	24.1	28.1
	Greedy + Node-Merging-Only	5.24	5.32	7255	3.1	24.1	27.1

Surprisingly, the Greedy method combined with Node-Merging-Only comes very close to the performance of the Dynamic-MSAH method, despite being much cheaper to build (see Sec. 5.3). As shown in Fig. 8, we observe that the Greedy + Node-Merging-Only method achieves comparable traversal performance to the Dynamic-MSAH + Merging method on all test cases. This tracks with our observation that the Greedy + Node-Merging-Only method produces similar MSAH and memory footprint (see Table 1). Both methods perform particularly well compared to the Baseline when memory access is incoherent (secondary and tertiary rays). This would indicate that the more memory-bound the traversal is, the more important these optimizations are.

Table 2. ***Traversal performance*** *in MRays/s, measured using Arches.*

		Primary	Reflections	Secondary	Tertiary
Crytek Sponza	Baseline	4096	2542	2084	1627
	Dynamic-SAH	4019	2861	1700	1272
	Dynamic-SAH + Merging	4339	3105	2530	1869
	Dynamic-MSAH	4373	3213	2644	1995
	Dynamic-MSAH + Merging	4372	3229	2678	1967
	Greedy	4049	3023	2521	1929
	Greedy + Node-Merging-Only	3991	3048	2553	1878
Intel Sponza	Baseline	4029	2604	1380	924
	Dynamic-SAH	4033	2318	1129	738
	Dynamic-SAH + Merging	4483	3020	1720	1162
	Dynamic-MSAH	4506	3195	1877	1307
	Dynamic-MSAH + Merging	4440	3248	1941	1356
	Greedy	4431	3118	1829	1268
	Greedy + Node-Merging-Only	4468	3140	1887	1291
Bistro	Baseline	2134	1110	687	564
	Dynamic-SAH	2024	1042	545	429
	Dynamic-SAH + Merging	2769	1680	1052	871
	Dynamic-MSAH	2840	1760	1025	843
	Dynamic-MSAH + Merging	2854	1782	1105	917
	Greedy	2763	1680	1000	867
	Greedy + Node-Merging-Only	2781	1697	987	855
San Miguel	Baseline	2260	1410	779	501
	Dynamic-SAH	2214	1271	633	404
	Dynamic-SAH + Merging	2695	1766	1045	675
	Dynamic-MSAH	2730	1873	1096	714
	Dynamic-MSAH + Merging	2756	1908	1142	752
	Greedy	2675	1814	1082	700
	Greedy + Node-Merging-Only	2699	1844	1111	716
Hairball	Baseline	1328	745	389	333
	Dynamic-SAH	1530	675	294	339
	Dynamic-SAH + Merging	2312	1662	961	1171
	Dynamic-MSAH	2427	1804	1135	1266
	Dynamic-MSAH + Merging	2436	1838	1127	1283
	Greedy	2386	1768	1088	1253
	Greedy + Node-Merging-Only	2403	1743	1121	1238

Note that our tests contain no memory access related to shading or texturing. Therefore, it is possible that these results underestimate the performance improvement, as—in typical applications—the traversal cores must contend with the memory traffic for other rendering-related tasks.

We have also evaluated our BVHs with 64B and 128B node blocks (Fig. 9). While 128B node blocks result in smaller trees and fewer traversal steps, the larger access granularity produces more total memory traffic and, thus, fails to outperform the 64B nodes. A similar tradeoff also exists with primitive blocks. Reducing primitive blocks to 64B, increases the vertex duplication, and, thereby, substantially increases the size of the tree. This, in turn, degrades performance. These two cases

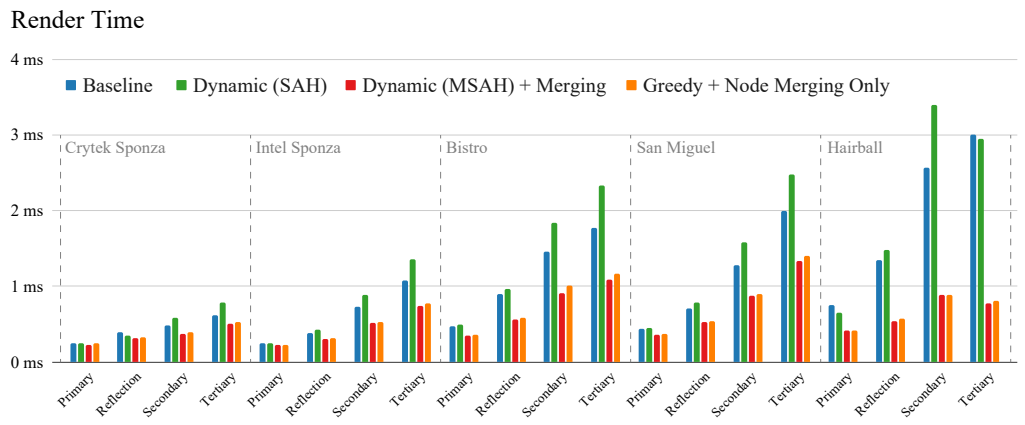


Fig. 8. *Comparison of frame time between the baseline (BVH used in Vaidyanathan et al. [2022]), our BVH using the existing build method, and our BVH using the proposed build methods. Simply introducing the new BVH without the build methods results in slightly worse performance. With the build methods, our BVH performs substantially better.*

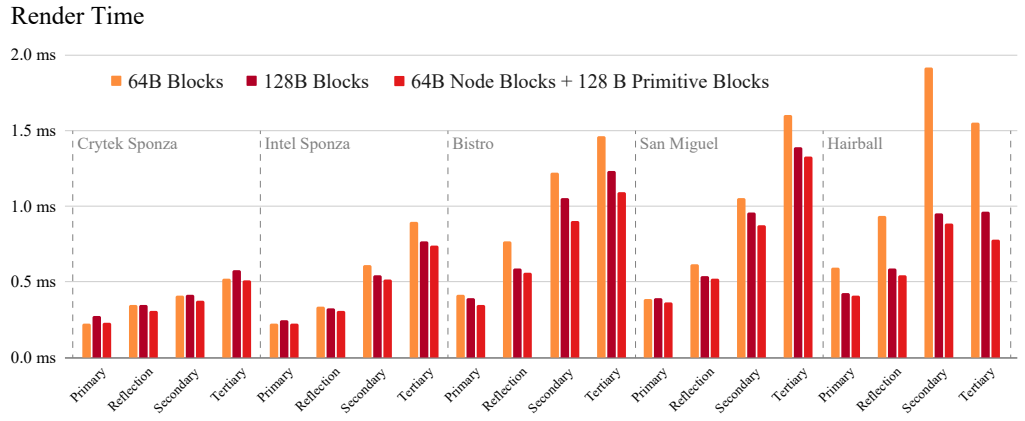


Fig. 9. *Comparison of using 64B and 128B for node blocks and primitive blocks. Using 64B blocks for both results in lower performance in all cases, except for primary rays. Using 128B blocks for both helps with the performance in larger scenes, but degrades the performance in smaller scenes. Using 64B node blocks and 128B primitive blocks uniformly performs better than the others.*

highlight the importance of balancing total memory footprint with memory traffic (MSAH), as both are important for high-performance ray traversal.

5.3 Build Times

We present build times of different methods with and without merging in Table 3 using a serial CPU implementation. All methods exhibit the same structure: an up pass for collapse, a down pass for collapse, and then a down pass for node merging and primitive merging. The notable exception is merging for the greedy build, where the Node-Merging-Only happens on the last level and, therefore, contains no dependencies.

Table 3. ***Build Times*** *in milliseconds measured on the CPU.*

		Leaf Collapse	Node Collapse	Node Merge	Leaf Merge	Total
Crytek Sponza	Dynamic-SAH	–	179	3	57	239
	Dynamic-MSAH	–	182	1	24	207
	Greedy	58	13	1	–	72
Intel Sponza	Dynamic-SAH	–	862	14	238	1114
	Dynamic-MSAH	–	774	4	58	836
	Greedy	186	63	5	–	260
Bistro	Dynamic-SAH	–	648	14	207	869
	Dynamic-MSAH	–	665	4	64	733
	Greedy	195	50	5	–	250
San Miguel	Dynamic-SAH	–	2597	51	762	3410
	Dynamic-MSAH	–	2362	14	187	2563
	Greedy	748	192	18	–	958
Hairball	Dynamic-SAH	–	613	8	138	759
	Dynamic-MSAH	–	573	3	20	596
	Greedy	181	53	4	–	238

In evaluating build performance, we see that the Dynamic-MSAH method slightly outperforms Dynamic-SAH. Most of this performance comes from cheaper leaf merging, which is a result of fuller primitive blocks with Dynamic-MSAH before merging.

Furthermore, we see that, as expected, the greedy collapse substantially outperforms both Dynamic methods in terms of build time. In these tests, our Greedy build method is 2.5 to 3.2 times faster than our Dynamic-MSAH build. In summary, our Greedy method is the fastest to build by a significant margin. Adding node merging on top of that only minimally increases the build time of the Greedy method.

While in practice, BVHs for hardware-accelerated ray traversal tend to be built on the GPU, we provide our CPU build numbers to give an idea of the relative performance between algorithms. In reality, these algorithms may scale better or worse on the GPU due to factors such as branch divergence, incoherent memory access, and thread synchronization. Due to the challenge of designing a high-performance GPU builder, we leave the implementation and evaluation of efficient GPU builders to future work.

6 Conclusion and Future Work

We present a new wide BVH encoding scheme for hardware traversal with a focus on improving performance by efficiently utilizing the memory hierarchy. We do this by modifying the traditional SAH metric, and then describe four methods for building higher-quality compressed wide BVHs that map well to the memory-access characteristics of traversal. We evaluate these methods using a cycle-level simulation with accurate memory system modeling and show that by combining these methods, various tradeoffs between build performance and traversal performance can be made. This work highlights the importance of considering memory system behavior as a primary concern for improving system performance, and of introducing semantics about the compression scheme directly into the build pipeline.

Future work should continue to focus on memory system performance and efficiently mapping these algorithms onto the GPU. In particular, the greedy method we propose should be an excellent candidate for real-time builds when combined with H-PLOC. Tightly integrating these two, similar

to what was done in [Barbier and Paulin 2025], could produce a GPU builder that is both fast and high-quality. Another interesting area of future work would be applying the dynamic method to DGF in order to produce SAH-friendly block packing. Finally, combining these techniques with spatial splits could produce even higher-quality BVHs. One of the major problems with spatial splits is the primitive data duplication; however, if the split boxes reference the same primitive block, then duplication is not necessary. Future work should look into augmenting this format to use spatial splits.

Acknowledgments

We thank David McAllister for supporting this project, which started at AMD. This work was supported in part by NSF Grant #2504980 and a gift from AMD.

References

Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)*, Rita Borgo and Wen Tang (Eds.). The Eurographics Association. doi:/10.2312/cgvc.20141206

Wilhem Barbier and Mathias Paulin. 2025. Fused Collapsing for Wide BVH Construction. *Computer Graphics Forum* (2025). doi:10.1111/cgf.70213

Joshua Barczak, Carsten Benthin, and David McAllister. 2024. DGF: A Dense, Hardware-Friendly Geometry Format for Lossily Compressing Meshlets with Arbitrary Topologies. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 46 (Aug. 2024), 17 pages. doi:10.1145/3675383

Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. 2022. PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 31 (July 2022), 13 pages. doi:10.1145/3543867

Carsten Benthin, Daniel Meister, Joshua Barczak, Rohan Mehalwal, John Tsakok, and Andrew Kensler. 2024. H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 30 (Aug. 2024), 14 pages. doi:10.1145/3675377

Jiří Bittner, Michal Hapala, and Vlastimil Havran. 2013. Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum* 32, 1 (2013), 85–100. doi:10.1111/cgf.12000

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum* 27, 4 (2008), 1225–1233. doi:10.1111/j.1467-8659.2008.01261.x

Leonardo R. Domingues and Helio Pedrini. 2015. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (Los Angeles, California) *(HPG '15)*. Association for Computing Machinery, New York, NY, USA, 13–20. doi:10.1145/2790060.2790065

Manfred Ernst and Gunther Greiner. 2008. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*. 35–40. doi:10.1109/RT.2008.4634618

Jacob Haydel, Gaurav Bhokare, Kunnong Zeng, Pengpei Hong, Sushant Kondguli, Brian Budge, Erik Brunvand, and Cem Yuksel. 2025. Arches: A Cycle Level Simulator for Exploring Massively Parallel Ray Tracing Architectures. *Computer Graphics Forum (HPG 2025)* 44, 8 (2025), 11 pages. doi:10.1111/cgf.70212

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* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 33–37.

Andrew Kensler. 2008. Tree Rotations for Improving Bounding Volume Hierarchies. *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, 73 – 76. doi:10.1109/RT.2008.4634624

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384. doi:10.1111/j.1467-8659.2009.01377.x

David J. MacDonald and Kellogg S. Booth. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput.* 6, 3 (May 1990), 153–166. doi:10.1007/BF01911006

Daniel Meister and Jiří Bittner. 2018. Parallel Reinsertion for Bounding Volume Hierarchy Optimization. *Computer Graphics Forum* 37, 2 (2018), 463–473. doi:10.1111/cgf.13376

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum* 40, 2 (2021), 683–712. doi:10.1111/cgf.142662

Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. 2022. Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 263–281. doi:10.1109/MICRO56248.2022.00027

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial splits in bounding volume hierarchies. *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)*, 7–13. doi:10.1145/1572769.1572771

Karthikeyan Vaidyanathan, Sven Woop, and Carsten Benthin. 2022. Wide BVH traversal with a short stack. In *Proceedings of the Conference on High-Performance Graphics* (Strasbourg, France) *(HPG '19)*. Eurographics Association, Goslar, DEU, 15–19. doi:10.2312/hpg.20191190

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs -. In *2008 IEEE Symposium on Interactive Ray Tracing*. 49–57. doi:10.1109/RT.2008.4634620

Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren A. Hunt, Steven G. Parker, and Peter Shirley. 2009. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum* 28 (2009). https://api.semanticscholar.org/CorpusID:889044

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. doi:10.1145/3105762.3105773