Fused Collapsing for Wide BVH Construction

Wilhem Barbier and Mathias Paulin

IRIT, Université de Toulouse, CNRS

Abstract
We propose a novel approach for constructing wide bounding volume hierarchies on the GPU by integrating a simple bottom-up collapsing procedure within an existing binary bottom-up BVH builder. Our approach directly constructs a wide BVH without traversing a temporary binary BVH as done by previous approaches and achieves $1.4 - 1.6\times$ lower build times. We demonstrate the ability of our algorithm to output compressed wide BVHs using existing compressed representations. We analyze the impact of our method on software raytracing performance and show that it reduces the overall frame time on complex dynamic scenes where rebuilding the BVH every frame is the limiting factor on rendering performance.

CCS Concepts
• *Computing methodologies* → *Ray tracing; Massively parallel algorithms;*

1. Introduction

The bounding volume hierarchy (BVH) is a widespread acceleration structure that enables raytracing queries to scale efficiently to large scenes. While the cost of constructing a BVH can be amortized over many rays for static scenes, for real-time raytracing of dynamic scenes the construction time is often as important as the tracing time for overall performance. This led to the development of efficient BVH construction algorithms which can take advantage of the GPU's massive parallelism for improved performance. However the existing literature for GPU algorithms focuses on constructing binary BVHs while the current state-of-the-art for software and hardware raytracing is the wide BVH [YKL17,BMB*24]: having more than two children per node results in shallower hierarchies thus lowering the latency of tree traversal, and allows for storing the sibling bounding boxes using a compressed representation to reduce the memory footprint. Several CPU algorithms for wide BVH construction have been published, but this remains a blind spot in the GPU literature with only a single published algorithm [BMB*24] to the best of our knowledge.

In this paper, we propose a fast GPU algorithm for wide BVH construction. The key idea of our method is to use a bottom-up collapsing procedure which allows us to fuse this collapsing step with a bottom-up binary BVH builder such as H-PLOC [BMB*24]. This *fused collapsing* procedure consistently outperforms state-of-the-art GPU collapsing algorithms, and leads to lower frame time when tracing few rays per pixel.

Our contributions are :

- a collapsing procedure which transforms a binary BVH into a wide BVH in a bottom-up manner,

- a fused implementation of this procedure within H-PLOC that does not require traversing a binary BVH.

2. Related Work

Binary BVH builders The problem of efficiently building a high-quality BVH has received a lot of attention from the graphics community. The surface area heuristic (SAH) is the most widely-used BVH quality metric as it is easy to compute and to optimize while being well-correlated with ray-tracing performance. On CPUs the top-down binned SAH builder introduced by Wald [Wal07] is the de-facto standard algorithm thanks to its speed and high-quality results. On GPUs the introduction of the LBVH algorithm [LGS*09, KA13, Ape14] was a major improvement over the previous state-of-the-art as it reduced the problem of BVH construction to integer sorting which admits efficient parallel algorithms [SJ17, AM22]. Later works focused on alleviating the poor quality of the resulting tree using a higher-quality builder for the first levels of the tree [PL10], or using a post-process pass on the BVH [KA13, DP15]. Meister and Bittner introduced the Parallel Locally-Ordered Clustering (PLOC) algorithm [MB17] which iteratively merges clusters of primitives and accelerates nearest-neighbor computation by restricting the search to a small set of candidates. This set of candidates is defined as a 1D neighborhood in the Morton-sorted array of clusters. This algorithm produces high-quality hierarchies while outperforming previous high-quality BVH builders and exploits the full parallelism of current GPUs. Benthin et al have improved the efficiency of the algorithm while keeping the same core loop: PLOC++ [BDTD22] proposes several optimizations such as splitting the primitives array into chunks that are processed independently by a thread group to reduce the
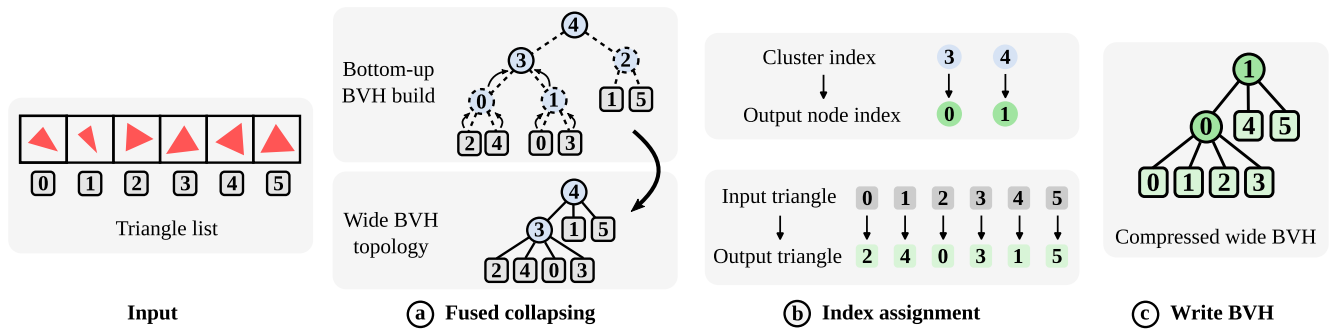
Figure 1: *Our wide BVH construction algorithm is divided in three steps: first we integrate a bottom-up collapsing algorithm within an existing binary bottom-up BVH builder, allowing us to directly compute the topology of our wide BVH without additional traversal. Then we assign indices so that sibling nodes and triangles are numbered consecutively, and finally we write the wide BVH using a compressed representation.*

number of kernel launches, and halving the number of distance computations using its commutative property. H-PLOC [BMB*24] builds the BVH in a single kernel launch: threads grow a set of input clusters using a bottom-up traversal of the LBVH hierarchy until the number of clusters exceeds a threshold, then all threads within the warp cooperate to build a subtree from these clusters using PLOC and resume their traversal. While top-down builders tend to produce hierarchies with higher raytracing throughput compared to bottom-up approaches such as PLOC, they are significantly harder to parallelize and existing top-down GPU algorithms have significantly lower build performance compared to state-of-the-art bottom-up builders [TDDB23].

Wide BVH construction There are two approaches to wide BVH construction: either build a wide BVH directly, or build a binary BVH first and convert it into a wide BVH as a second step. In the first category, Wald et al. [WBB08] propose to adapt the widespread top-down binned SAH builder by repeatedly splitting the children of a node until it is filled. The same paper also describes a collapsing algorithm to convert a binary BVH into a wide BVH [WBB08] using three operations: merging an interior node into its parent, merging two leaf nodes together and merging two interior nodes together. Ernst and Greiner [EG08] as well as Dammertz et al. [DHK08] build a *K*-wide BVH by keeping every *k*-th level of the binary BVH, where $k = \log_2 K$. Pinto [Pin10] and Ylitie et al. [YKL17] use a dynamic programming approach to compute the SAH-optimal wide BVH from a binary BVH.

Wide BVHs on the GPU On GPUs, Guthe [Gut14] found that tracing a binary hierarchy was latency-bound on their hardware and that using a 4-wide BVH increases tracing performance. Ylitie et al. [YKL17] show that their compressed wide BVH consistently outperforms previous works for software raytracing queries. By quantizing the coordinates of child bounding boxes to a single byte they reduce the size of an 8-wide node to 80 bytes. They also introduce a compressed stack which reduces costly global memory accesses during traversal, along with other optimizations to reduce thread divergence. Lier et al. [LSS18] take inspiration from the SIMD traversal algorithms used on CPUs and distribute the

child bounding boxes of a node over multiple lanes so they can be intersected in parallel. Vaidyanathan et al. [VWB19] generalize the binary restart trail of Laine [Lai10] to wide BVHs, leading to reduced stack usage with a simple traversal algorithm that is suitable for hardware implementation. While the current state-of-the-art structure for GPU raytracing is the wide BVH, there has been surprisingly little research on building these hierarchies on the GPU. To our knowledge the only published algorithm was introduced by Benthin et al [BMB*24]: their algorithm traverses the input binary BVH top-down and repeatedly fills all slots of a wide node by replacing the child with largest area by grand-children, similarly to Wald [WBB08].

While converting a binary BVH to a wide BVH allows us to reuse existing build algorithms, the collapsing step is expensive as it must traverse the binary BVH which will then be discarded. In practice this step is often more expensive than the binary hierarchy construction with a fast build algorithm [BMB*24, MKVH24]. To alleviate this issue, we propose a wide BVH construction algorithm that avoids this additional traversal by fusing a simple collapsing procedure with an existing high-performance binary BVH builder such as H-PLOC.

3. Fused bottom-up collapsing

In this section we present our wide BVH construction algorithm, which takes as input a sorted list of triangles and produces two outputs: the first is a compressed wide BVH, and the second is a permutation of triangle IDs such that all triangles referenced by a node have consecutive indices. The algorithm consists of three stages, each implemented with a separate kernel launch (Figure 1):

- First, fused collapsing outputs both an array of clusters and the topology of our wide BVH (sections 3.1,3.2). We begin by describing a bottom-up collapsing procedure that operates on a binary BVH to produce a wide BVH (section 3.1). We then explain how to integrate this procedure within the PLOC and H-PLOC binary BVH builders (section 3.2) so that the binary BVH does not need to be explicitly stored and traversed.
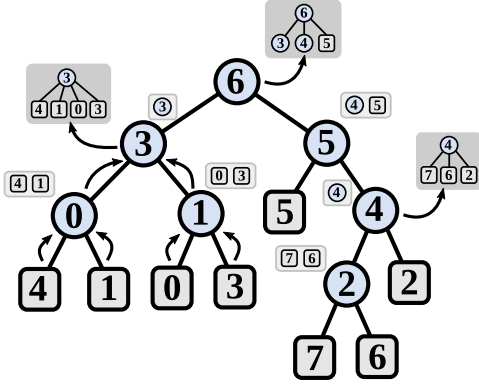- Second, we assign indices to every node of the wide BVH so

Figure 2: *Bottom-up collapsing procedure for a 4-wide BVH. Every node is labelled with a set of references, and a new wide node is created when the size of this set exceeds a threshold (equal to 2 here).*

- that siblings have consecutive indices. We also compute triangle IDs such that all triangles referenced by a node have consecutive indices (section 3.3).
- Third, we compute and write the compressed nodes of the output BVH (section 3.3).

3.1. Bottom-up collapsing

We describe here our collapsing algorithm as a stand-alone procedure that takes as input a binary BVH and constructs a *K*-wide BVH. The output of this procedure is the topology of the wide BVH, where every wide node is represented as the index of its corresponding node in the input BVH along with the *K* (or less) indices of its children in the input BVH. As written in Procedure 1, the input BVH is traversed bottom-up and during this traversal every node is labelled with a set *refs* of at most $\frac{K}{2}$ references to primitives or child nodes. For leaf nodes we initialize the references to the set of primitives contained by the node (lines 2-4). For internal nodes, we denote *R* the union of the references of its two children. We distinguish between two cases depending on the size of *R*:

- If $|R| \leq \frac{K}{2}$, we use *R* as the set of references for the current node (lines 7-9).
- Otherwise, we create a new node in the output wide BVH with its children given by the set *R* which contains at most *K* references. Then we set the references for the current node to a singleton containing its own index (lines 9-12).

The CREATEWIDENODE procedure increments an atomic counter to obtain an index into the wide nodes array, and writes out the binary node index and the references for the new wide node at this location. The wide node arrays thus defines the topology of the wide BVH while the bounding box of each wide node can be fetched from the corresponding binary node.

3.2. Fused collapsing

In this section we show how to integrate the bottom-up collapsing procedure into the PLOC and H-PLOC build algorithms.

Procedure 1: Bottom-up collapsing

1 Input *T : binary BVH, K: wide BVH arity*

2 foreach *leaf node n* **in** *T* **do**
3 | n.refs ← n.primitiveIDs;
4 end
5 foreach *internal node n* **in** *bottom-up traversal of T* **do**
6 | R ← n.leftChild.refs ∪ n.rightChild.refs;
7 | **if** $|R| \leq \frac{K}{2}$ **and** n ≠ T.root **then**
8 | | n.refs ← R;
9 | **else**
10 | | CREATEWIDENODE(n.index, R);
11 | | n.refs ← { n.index };
12 | **end**
13 end

PLOC algorithm To keep the description of our algorithm self-contained, we give a quick summary of the PLOC algorithm here and we defer to the paper [MB17] for more details. PLOC computes a bottom-up clustering of the input primitives, where each cluster corresponds to a node in the binary BVH. A cluster is described by its bounding box and two indices to its children. PLOC is an iterative algorithm, where every iteration creates new clusters and updates a list of active cluster indices. The algorithm initializes one cluster per primitive and repeats the following three stages until a single active cluster is left:

- During the nearest-neighbor stage, all active clusters iterate through their neighboring clusters according to Morton code and find their neighbor that minimizes the surface area of the merged bounding box.
- During the merging stage, pairs of clusters that are each other's nearest neighbor are merged together: a new cluster is created, and one of the two cluster indices is replaced by the new cluster's index while the other is marked as inactive.
- The compaction stage removes the inactive clusters indices from the list.

Fusing with PLOC To fuse the bottom-up collapsing procedure with PLOC we modify the definition of a cluster: instead of two children indices, clusters store a set of at most $\frac{K}{2}$ references. We initialize the reference set of initial clusters to a singleton containing their cluster index. Only the merging stage needs to be changed and we show the necessary modifications in Procedure 2 (highlighted in blue). These modifications are a straightforward adaptation of the collapsing procedure using the fact that a cluster in Procedure 2 directly corresponds to a binary node in Procedure 1. Since we do not store binary children indices for the clusters our modified PLOC does not result in a binary BVH, but instead produces an array of clusters and the topology of the wide BVH. This results in some memory overhead since our clusters store $\frac{K}{2}$ references instead of 2 for binary PLOC. Additionally, the topology of the wide BVH is stored separately using *K* integers per node of the wide BVH.

Fusing with H-PLOC H-PLOC is a variant of PLOC that builds the BVH in a single kernel dispatch, and is currently the state-of-the-art for GPU binary BVH builders [BMB*24]. H-PLOC can be

---

Procedure 2: PLOCMerge with fused collapsing

---

1   **if** *NN[NN[tidx]] == tidx* **and** *tidx < NN[tidx]* **then**
2   clusterIdx ← ATOMICADD(clusterCounter, 1);
3   C_0 ← clusters[clusterIndices[tidx]];
4   C_1 ← clusters[clusterIndices[NN[tidx]]];
5   aabb ← MERGE(C_0.box, C_1.box);
6   R ← C_0.refs $\cup$ C_1.refs;
7   **if** $|R| > \frac{K}{2}$ **or** *clusterIdx = 2× numPrims-2* **then**
8   wideIdx ← ATOMICADD(wideCounter, 1);
9   wideNodes[wideIdx] ← (clusterIdx, R);
10   R ← {clusterIdx};
11   **end**
12   clusterIndices[tidx] ← clusterIdx;
13   clusterIndices[NN[tidx]] ← −1;
14   clusters[clusterIdx] ← (aabb, R);
15 **end**

---

decomposed into an outer loop where every thread grows a set of input clusters based on their Morton code ranges until the number of clusters exceeds a threshold, and an inner loop where threads within a warp cooperate to build a subtree from this set of input clusters using PLOC. At the start of the inner loop, the bounding boxes of the cluster set are loaded from global memory into shared memory. We modify this loading step to fetch the cluster references and store them in shared memory so that they can be accessed during the merging stage. Similarly to PLOC, the merging stage is modified to compute the set of references of the merged cluster and optionally create a wide node (Procedure 2).

Merge penalty Fusing the collapsing algorithm into the binary BVH builder not only avoids an additional traversal, but also allows us to improve the quality of the wide BVH by nudging the build algorithm towards trees that the collapsing algorithm handles well. To do so we introduce a *merge penalty*: after computing the distance between two clusters during the nearest-neighbor phase, we multiply it by a fixed factor $\alpha \geq 1$ if both clusters do not have the same number of references. This heuristic biases the PLOC builder towards clusters with a power-of-two number of references and results in wide hierarchies with more children per node and lower SAH, especially in the case of 4-wide hierarchies. We determine experimentally that setting $\alpha = 1.3$ works well on average.

Output data The output of the fused collapsing stage is both a cluster array and the topology of the wide BVH. The wide BVH topology is represented as an array of wide nodes where each node stores the index of its corresponding cluster, which we will refer to as its *cluster index*, and the cluster indices of its children. The bounding boxes of clusters referenced by the wide BVH topology will be copied into the final compressed wide BVH (section 3.3).

3.3. Compressed wide BVH

Compressed wide BVHs are the state-of-the-art acceleration structure for GPU raytracing, therefore we want our construction algo-

rithm to output the nodes in a compressed format. We use a representation similar to the one described by Ylitie et al. [YKL17]:

- Every node stores its own bounding box using a 32-bit floating-point origin and 8-bit scale per dimension.
- Every node stores its child bounding boxes quantized to 8 bits per dimension.
- The children of a node have consecutive indices, as well as the triangle it references. This allows us to store a single child base index and triangle base index per node, along with bitmasks to specify which slots contain internal nodes and which contain triangles.

Index assignment phase We cannot use the index of a node in the wide nodes array as its index in the compressed BVH since it does not satisfy the condition that all siblings have consecutive indices. Therefore, we introduce an index assignment stage to compute an *output index* for every wide node. During this stage, wide nodes assign output indices to their children. Since wide nodes only have access to the cluster indices of their children, the output of this stage is a mapping from cluster index to output index. We dispatch one thread per wide node, which will reserve a range of output indices by counting the number of internal children and incrementing an atomic counter. Then it iterates over the children and stores the mapping from their cluster index to their output index. Similarly, it counts the number of triangles referenced by the node, reserves a range of indices with another atomic counter and stores them in a mapping from input triangle index to output triangle index. We store the base child index and base triangle index for every wide node to avoid some dependent memory accesses in the next stage.

Write BVH phase Finally we write out the compressed BVH: we again spawn one thread per wide node and, using the cluster index of the wide node, every thread fetches its bounding box from the cluster array and its output index from the previously computed mapping. Then it iterates over its children, reads their bounding box from the cluster array and computes their quantized representation. Finally, the child base index and triangle base index are copied into the compressed node which is written at the output index.

3.4. Partial reordering

The ordering of nodes can have a significant impact on raytracing performance depending on the size of the node structure, the scene, and the raytracing traversal algorithm. Unfortunately, since our indexing is non-deterministic as it depends on the order of execution of the atomic operations during the index assignment stage, we can not guarantee that the compressed wide BVH node ordering is suitable for efficient ray traversal. While it would be possible to reorder the whole hierarchy breadth-first as a post-process, this would require a traversal of the whole tree to compute the output indices. Instead we propose to reindex only the nodes of the first levels of the wide BVH, which are accessed very often and for which spatial locality is most critical to performance.

We implement this reordering procedure (Procedure 3) with a parallel breadth-first traversal of the first levels of the wide BVH using a single threadblock, which allows us to store a queue in shared memory and to synchronize the threads within the block.

This queue stores pairs of indices, where the first element is the input index of the node to be processed, and the second element is the index in the reordered tree. We process one level of the tree per iteration, with the number of active threads equal to the queue size: every active thread reads an index pair from the queue (line 7), then the queue is reset and the active threads fetch the corresponding node and reserve children indices by incrementing an atomic counter (lines 9-14). Active threads then reserve space in the queue for their children (line 15), and write index pairs for their children to the queue if it has enough capacity (lines 15-19). Finally, active threads update their node with the reordered child index and write it at the output location (lines 23-28). The loop ends when a thread cannot write its children into the queue. In our implementation every thread processes three items from the queue for $K = 4$ or two for $K = 8$. We use a threadblock of size 1024 which is the hardware limit for our GPU, which results in a queue capacity of 2048 or 3072 depending on K.

To ensure that this reordering does not write over existing nodes, we initialize the atomic counter used during the index assignment stage (section 3.3) to a constant m so that the first m nodes are vacant and can be used for the reordered tree. Taking m equal to twice the queue capacity guarantees that all new nodes will have an index lesser than m, thus avoiding any conflicts.

3.5. Implementation details

Integer set representation We represent the set of references with an integer vector of size $\frac{K}{2}$, using an invalid value to represent empty slots. The procedure to compute the union of two vectors a and b replaces invalid values of a with values from b until either b is empty or a is full. Since vectors stored in registers cannot be indexed dynamically, we compute a 6-bit tag from the size of both sets and handle the different cases with a switch statement on this tag. For example if both a and b have two elements with $\frac{K}{2} = 4$, the computed tag is (2 << 3) | 2 and the corresponding switch case replaces the last two (unused) values of a with the first two values from b.

Triangle reordering Along with the compressed wide BVH, our construction pipeline outputs a mapping from input to output triangle indices. We apply this mapping to reorder the triangle data as an additional step after the construction. Since this triangle reordering step is also needed when using top-down collapsing, we measure it separately from the hierarchy construction.

Node layout We use our construction algorithm to build 4-wide and a 8-wide compressed hierarchies. The 8-wide nodes use a similar representation to Ylitie et al's compressed wide BVH [YKL17] with 80 bytes per node. We adapt this representation to 4-wide nodes using 48 bytes per node.

4. Results

Experimental setting We compare H-PLOC with fused collapsing against Benthin et al.'s top-down collapsing algorithm [BMB*24], which takes as input a binary BVH. Comparing against H-PLOC+TOPDOWN allows us to understand the difference in build time and BVH quality due to the collapsing algorithm.

Procedure 3: Partial reordering

1 **Input** *tidx: thread idx, nodes: compressed wide nodes array*

2 **shared uint2** queue[] $\leftarrow$ {(rootIdx, 0)};

3 **shared uint** queueSize $\leftarrow$ 1;

4 **shared uint** nodeCounter $\leftarrow$ 0;

5 **while** *queueSize* $\neq$ 0 **and** *queueSize* $\leq$ *CAPACITY* **do**

6 　　active $\leftarrow$ tidx < queueSize;

7 　　**if** *active* **then**

8 　　　　indexPair $\leftarrow$ queue[tidx];

9 　　**end**

10 　　SYNCTHREADS();

11 　　queueSize $\leftarrow$ 0;

12 　　SYNCTHREADS();

13 　　**if** *active* **then**

14 　　　　node $\leftarrow$ nodes[indexPair.input];

15 　　　　outBaseIdx $\leftarrow$ ATOMICADD(nodeCounter, node.numChildren);

16 　　　　queueIdx $\leftarrow$ ATOMICADD(queueSize, node.numChildren);

17 　　　　**if** *queueIdx + node.numChildren* $\leq$ *CAPACITY* **then**

18 　　　　　　**for** *i = 0* **to** *node.numChildren-1* **do**

19 　　　　　　　　queue[queueIdx+i] $\leftarrow$ (node.childBaseIdx+i, outBaseIdx+i);

20 　　　　　　**end**

21 　　　　**end**

22 　　**end**

23 　　SYNCTHREADS();

24 　　**if** *active* **then**

25 　　　　**if** *queueSize* $\leq$ *CAPACITY* **then**

26 　　　　　　node.childBaseIdx $\leftarrow$ outBaseIdx;

27 　　　　**end**

28 　　　　nodes[indexPair.output] $\leftarrow$ node;

29 　　**end**

30 **end**

We also compare against LBVH+TOPDOWN since we also aim to enable fast rebuilds for dynamic scenes at the cost of some tracing speed. All experiments were performed on a NVIDA GeForce RTX 3090 GPU and an Intel Xeon Silver 4214R CPU @ 2.40GHz. We use our own implementation for the LBVH builder and top-down collapsing, and we use the H-PLOC implementation from the HIPRT repository [MKVH24]. Both LBVH and H-PLOC use 64-bit Morton codes. To evaluate the tracing performance of the BVHs we use a software raytracing kernel based on the algorithm by Ylitie et al [YKL17]. We use child sorting for front-to-back traversal, since we found it to perform better than the octant traversal method and it allows us to use the same code for BVH4 and BVH8 traversal. We trace a primary ray and a diffuse secondary ray per pixel at 1920×1080 resolution, and measure the tracing time for the secondary ray. We use cosine sampling to compute the direction of the diffuse ray to measure tracing performance of incoherent rays. We measured the build time, trace time, SAH and number of traversed

Wilhem Barbier & Mathias Paulin / Fused Collapsing for Wide BVH Construction

	Algorithm	Hierarchy (ms)	Build (ms)	SAH	Trace (ms)	Avg traversed #nodes	Combined time (ms)
		Bistro Exterior (2.8M triangles)					
BVH4	H-PLOC+TopDown	4.9 (×1)	6.9 (×1)	70.3 (×1)	7.9 (×1)	53.1 (×1)	14.8 (×1)
BVH4	LBVH+TopDown	3.9 (×0.80)	5.9 (×0.86)	80.5 (×1.14)	10.3 (×1.32)	64.4 (×1.21)	16.3 (×1.10)
BVH4	Ours	2.8 (×0.57)	4.6 (×0.67)	68.9 (×0.98)	10.9 (×1.39)	57.0 (×1.07)	15.6 (×1.05)
BVH8	H-PLOC+TopDown	4.8 (×1)	6.8 (×1)	57.9 (×1)	7.8 (×1)	35.0 (×1)	14.6 (×1)
BVH8	LBVH+TopDown	4.1 (×0.84)	6.0 (×0.88)	63.5 (×1.10)	8.8 (×1.14)	41.7 (×1.19)	14.9 (×1.02)
BVH8	Ours	2.9 (×0.59)	4.7 (×0.69)	56.9 (×0.98)	9.4 (×1.21)	38.0 (×1.09)	14.1 (×0.97)
		Hairball (2.9M triangles)					
BVH4	H-PLOC+TopDown	4.4 (×1)	6.5 (×1)	350.3 (×1)	10.6 (×1)	48.5 (×1)	17.1 (×1)
BVH4	LBVH+TopDown	3.9 (×0.88)	5.9 (×0.91)	363.7 (×1.04)	8.4 (×0.80)	49.2 (×1.01)	14.4 (×0.84)
BVH4	Ours	2.5 (×0.55)	4.4 (×0.68)	346.1 (×0.99)	8.8 (×0.83)	53.0 (×1.09)	13.2 (×0.77)
BVH8	H-PLOC+TopDown	4.9 (×1)	7.0 (×1)	295.0 (×1)	8.9 (×1)	34.7 (×1)	15.9 (×1)
BVH8	LBVH+TopDown	4.1 (×0.84)	6.2 (×0.88)	295.6 (×1.00)	8.1 (×0.91)	34.1 (×0.98)	14.3 (×0.90)
BVH8	Ours	2.5 (×0.52)	4.5 (×0.64)	284.3 (×0.96)	8.6 (×0.96)	37.8 (×1.09)	13.1 (×0.82)
		Rungholt (5.8M triangles)					
BVH4	H-PLOC+TopDown	8.6 (×1)	12.9 (×1)	64.9 (×1)	2.0 (×1)	21.8 (×1)	14.9 (×1)
BVH4	LBVH+TopDown	7.9 (×0.92)	12.2 (×0.94)	78.0 (×1.20)	2.0 (×1.02)	22.9 (×1.05)	14.2 (×0.95)
BVH4	Ours	4.5 (×0.52)	8.5 (×0.66)	72.4 (×1.12)	2.4 (×1.21)	26.5 (×1.22)	10.9 (×0.73)
BVH8	H-PLOC+TopDown	8.3 (×1)	12.5 (×1)	47.1 (×1)	2.1 (×1)	15.3 (×1)	14.6 (×1)
BVH8	LBVH+TopDown	7.8 (×0.94)	11.9 (×0.95)	54.9 (×1.16)	2.1 (×1.03)	16.0 (×1.04)	14.1 (×0.97)
BVH8	Ours	4.6 (×0.56)	8.7 (×0.69)	56.2 (×1.19)	2.7 (×1.31)	20.3 (×1.33)	11.4 (×0.78)
		San Miguel (10M triangles)					
BVH4	H-PLOC+TopDown	16.1 (×1)	22.7 (×1)	40.8 (×1)	5.2 (×1)	43.3 (×1)	27.9 (×1)
BVH4	LBVH+TopDown	13.4 (×0.83)	19.9 (×0.87)	53.9 (×1.32)	7.0 (×1.35)	63.0 (×1.46)	26.8 (×0.96)
BVH4	Ours	8.8 (×0.54)	14.8 (×0.65)	42.9 (×1.05)	6.5 (×1.26)	48.8 (×1.13)	21.3 (×0.76)
BVH8	H-PLOC+TopDown	16.5 (×1)	22.9 (×1)	31.1 (×1)	5.3 (×1)	29.9 (×1)	28.2 (×1)
BVH8	LBVH+TopDown	13.6 (×0.83)	19.9 (×0.87)	38.8 (×1.25)	6.9 (×1.30)	41.7 (×1.40)	26.8 (×0.95)
BVH8	Ours	9.1 (×0.55)	15.1 (×0.66)	34.1 (×1.10)	6.8 (×1.28)	38.0 (×1.27)	22.0 (×0.78)
		Powerplant (12.8M triangles)					
BVH4	H-PLOC+TopDown	19.1 (×1)	27.4 (×1)	25.8 (×1)	10.3 (×1)	54.4 (×1)	37.7 (×1)
BVH4	LBVH+TopDown	17.0 (×0.89)	25.2 (×0.92)	31.2 (×1.21)	12.5 (×1.22)	79.1 (×1.45)	37.8 (×1.00)
BVH4	Ours	9.7 (×0.51)	17.3 (×0.63)	25.3 (×0.98)	10.0 (×0.97)	57.9 (×1.07)	27.3 (×0.72)
BVH8	H-PLOC+TopDown	19.9 (×1)	28.0 (×1)	20.9 (×1)	10.0 (×1)	38.4 (×1)	38.0 (×1)
BVH8	LBVH+TopDown	17.5 (×0.88)	25.5 (×0.91)	23.9 (×1.15)	11.6 (×1.16)	51.6 (×1.34)	37.1 (×0.98)
BVH8	Ours	10.1 (×0.51)	17.6 (×0.63)	20.9 (×1.00)	11.6 (×1.16)	44.4 (×1.16)	29.2 (×0.77)
		Moore Lane House (15.2M triangles)					
BVH4	H-PLOC+TopDown	24.3 (×1)	34.6 (×1)	16.2 (×1)	4.0 (×1)	32.7 (×1)	38.5 (×1)
BVH4	LBVH+TopDown	20.6 (×0.85)	30.7 (×0.89)	18.6 (×1.14)	4.6 (×1.16)	40.5 (×1.24)	35.3 (×0.92)
BVH4	Ours	13.1 (×0.54)	22.6 (×0.65)	17.7 (×1.09)	4.5 (×1.14)	36.9 (×1.13)	27.0 (×0.70)
BVH8	H-PLOC+TopDown	25.2 (×1)	35.2 (×1)	11.9 (×1)	4.1 (×1)	22.6 (×1)	39.3 (×1)
BVH8	LBVH+TopDown	21.1 (×0.84)	31.0 (×0.88)	13.2 (×1.11)	4.7 (×1.13)	27.1 (×1.20)	35.7 (×0.91)
BVH8	Ours	13.4 (×0.53)	22.8 (×0.65)	13.9 (×1.17)	5.1 (×1.23)	28.6 (×1.26)	27.9 (×0.71)

Table 1: *Performance comparison between our fused collapsing and top-down collapsing (*H-PLOC *and* LBVH*) on static scenes. All results are averaged over six viewpoints per scene. The **Build** column measures the whole build process with Morton code computation, sorting, hierarchy construction and triangle reordering. The **Hierarchy** column only measures the hierarchy construction step.*

	Algorithm	Hierarchy (ms)	Build (ms)	SAH	Trace (ms)	Avg traversed #nodes	Combined time (ms)
		Breaking Lion (1.6M triangles average)					
BVH4	H-PLOC+TopDown	2.8 (×1)	4.0 (×1)	160.9 (×1)	1.4 (×1)	38.5 (×1)	5.5 (×1)
BVH4	LBVH+TopDown	2.1 (×0.77)	3.4 (×0.84)	180.4 (×1.12)	1.7 (×1.15)	46.4 (×1.20)	5.0 (×0.92)
BVH4	Ours	1.8 (×0.66)	2.9 (×0.71)	161.6 (×1.00)	1.6 (×1.11)	41.9 (×1.09)	4.5 (×0.81)
BVH8	H-PLOC+TopDown	2.9 (×1)	4.3 (×1)	130.0 (×1)	1.5 (×1)	25.5 (×1)	5.8 (×1)
BVH8	LBVH+TopDown	2.2 (×0.76)	3.4 (×0.80)	141.3 (×1.09)	1.7 (×1.11)	30.1 (×1.18)	5.1 (×0.88)
BVH8	Ours	1.9 (×0.67)	2.9 (×0.69)	133.2 (×1.02)	1.8 (×1.17)	30.1 (×1.18)	4.7 (×0.81)
		Flooded Sponza (4.8M triangles average)					
BVH4	H-PLOC+TopDown	7.5 (×1)	11.0 (×1)	54.3 (×1)	4.0 (×1)	37.9 (×1)	15.0 (×1)
BVH4	LBVH+TopDown	6.1 (×0.81)	9.6 (×0.88)	63.0 (×1.16)	4.4 (×1.09)	42.1 (×1.11)	14.0 (×0.93)
BVH4	Ours	4.3 (×0.58)	7.4 (×0.68)	57.5 (×1.06)	4.7 (×1.19)	42.9 (×1.13)	12.1 (×0.81)
BVH8	H-PLOC+TopDown	7.7 (×1)	11.1 (×1)	40.8 (×1)	4.1 (×1)	26.0 (×1)	15.3 (×1)
BVH8	LBVH+TopDown	6.3 (×0.81)	9.7 (×0.87)	45.6 (×1.12)	4.5 (×1.08)	28.6 (×1.10)	14.1 (×0.93)
BVH8	Ours	4.5 (×0.58)	7.5 (×0.68)	45.6 (×1.12)	5.2 (×1.25)	32.4 (×1.24)	12.7 (×0.83)
		Falling Bunnies (5.3M triangles average)					
BVH4	H-PLOC+TopDown	8.4 (×1)	12.2 (×1)	9.5 (×1)	2.4 (×1)	23.8 (×1)	14.6 (×1)
BVH4	LBVH+TopDown	6.7 (×0.80)	10.4 (×0.86)	12.3 (×1.29)	3.2 (×1.29)	32.2 (×1.35)	13.6 (×0.93)
BVH4	Ours	5.0 (×0.60)	8.4 (×0.69)	10.7 (×1.13)	2.9 (×1.18)	27.6 (×1.16)	11.2 (×0.77)
BVH8	H-PLOC+TopDown	8.5 (×1)	12.2 (×1)	6.8 (×1)	2.4 (×1)	16.3 (×1)	14.6 (×1)
BVH8	LBVH+TopDown	7.1 (×0.83)	10.7 (×0.88)	8.4 (×1.24)	3.0 (×1.25)	21.1 (×1.29)	13.7 (×0.94)
BVH8	Ours	5.2 (×0.61)	8.5 (×0.70)	8.1 (×1.20)	3.1 (×1.27)	20.9 (×1.28)	11.6 (×0.79)
		Splash (6.9M triangles average)					
BVH4	H-PLOC+TopDown	11.1 (×1)	16.2 (×1)	21.8 (×1)	2.4 (×1)	29.1 (×1)	18.6 (×1)
BVH4	LBVH+TopDown	9.4 (×0.85)	14.4 (×0.89)	23.2 (×1.06)	2.5 (×1.01)	27.9 (×0.96)	16.8 (×0.90)
BVH4	Ours	6.1 (×0.55)	10.6 (×0.65)	24.2 (×1.11)	3.1 (×1.28)	33.4 (×1.15)	13.7 (×0.74)
BVH8	H-PLOC+TopDown	10.3 (×1)	15.2 (×1)	15.4 (×1)	2.3 (×1)	19.8 (×1)	17.6 (×1)
BVH8	LBVH+TopDown	8.9 (×0.86)	13.7 (×0.90)	16.3 (×1.06)	2.2 (×0.96)	19.0 (×0.96)	15.9 (×0.91)
BVH8	Ours	6.2 (×0.61)	10.7 (×0.70)	18.5 (×1.20)	3.2 (×1.37)	25.4 (×1.28)	13.9 (×0.79)

Table 2: *Performance comparison between our fused collapsing and top-down collapsing (H-PLOC and LBVH) on dynamic scenes. All results are averaged over the whole animation.*

nodes for a set of dynamic scenes (Table 2) and static scenes (Table 1). All scenes are shown in the accompanying video.

Build time Building a compressed wide BVH from a list of triangles consists of many steps: Morton code computation, sorting, hierarchy construction and triangle reordering. We use triangles as our primitives instead of triangle pairs which are used in the H-PLOC paper. Since we only change the hierarchy construction step, we measure it separately in the **Hierarchy** column. However to be representative of a use-case in a real scenario, we also measure the time for the whole build process in the **Build** column. Our algorithm is consistently faster than both H-PLOC+TopDown and LBVH+TopDown for hierarchy construction (33%-49% speedup), which leads to reduced overall build times (31%-37% speedup). We break down the hierarchy construction time for Ours and H-PLOC+TopDown in Figure 3: top-down collapsing takes up most of the hierarchy construction time of H-PLOC+TopDown, and our approach achieves significant

speedups by avoiding this additional traversal. The fused collapsing stage is the main bottleneck of our algorithm, and is slightly slower than the binary H-PLOC builder due to reduced occupancy and the overhead of creating wide nodes. We present a more comprehensive breakdown of the construction time in the supplementary material.

SAH Since our bottom-up collapsing does not take node areas into account, it tends to result in a higher SAH than top-down collapsing which greedily minimizes child areas (+13% at worst for BVH4, +20% for BVH8). For 4-wide hierarchies our approach achieves similar SAH to TopDown on most scenes, thanks to the fact that the collapsing is more constrained and that our merge penalty is more effective on 4-wide hierarchies.

Trace time Our algorithm overall increases the tracing time compared to H-PLOC+TopDown which is expected as our objective is to enable fast wide BVH construction at the cost of some tracing performance. This increase in tracing time is very scene-dependent,
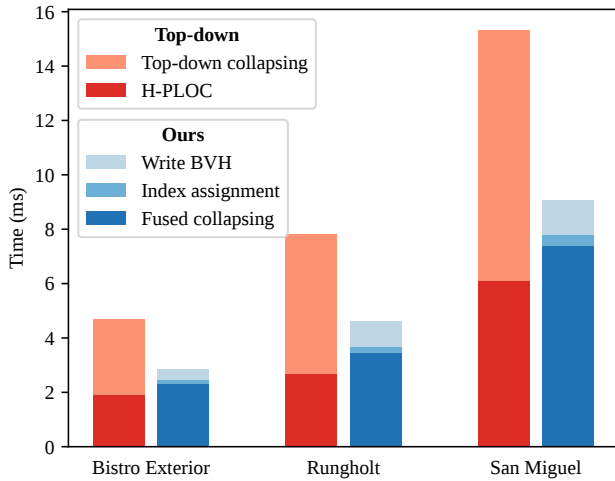
Figure 3: *Breakdown of the hierarchy construction cost for* OURS *and* H-PLOC+TOPDOWN *for a 4-wide BVH*

	Trace	SAH	Avg traversed	Max traversed
BVH4	−12%	−7%	−10%	−22%
BVH8	−1%	−3%	−3%	−2%

Table 3: *We measure the effect of our merge penalty on the trace time, SAH, and number of traversed nodes per ray (average and maximum) on our test scenes. We show here the relative difference for these measurements, averaged over the scenes (lower is better).*

up to a 39% increase for the *Bistro Exterior* scene and an average of +13% for the BVH4 and +22% for the BVH8 on other scenes.

Combined time To estimate the overall performance impact of our method over TOPDOWN, we compute the combined time by summing the **Build** and **Trace** columns. Our algorithm achieves a speedup of 18-29% on all scenes except *Bistro Exterior*, where it is 5% slower. In general larger scenes benefit more from our algorithm as tracing time is less sensitive to the number of primitives than the build time. We repeat the experiments with 4 and 8 incoherent secondary rays per pixel and show the results in the supplementary material. Our algorithm still achieves lower combined time on all scenes except *Bistro Exterior* at 4spp, and for 8spp our approach still performs better on large scenes such as *Powerplant* and *Moore Lane House*.

Tree quality and merge penalty We first examine the impact of the merge penalty on the topology of the resulting BVH. Table 4 compares the average number of children per node using top-down collapsing against our bottom-up algorithm with and without merge penalty. It shows that bottom-up collapsing achieves better slot utilization for both BVH4 and BVH8, and that the merge penalty produces fuller trees with fewer nodes. However this single statistic does not tell the whole story, as the top-down and bottom-up approaches produce trees with different characteristics. The first row of Figure 4 shows that top-down collapsing results in slot utiliza-

	Children / node		Num nodes	
	BVH4	BVH8	BVH4	BVH8
Top-down	3.1	4.1	100%	100%
Ours, $\alpha = 1$	3.4	5.9	87%	63%
Ours, $\alpha = 1.3$	3.7	6.4	76%	56%

Table 4: *Comparison of the average number of children per node and number of nodes between top-down collapsing, bottom-up collapsing without merge penalty and bottom-up collapsing with merge penalty.*

tion that is very high for the first levels of the tree and decreases as the nodes get deeper. Inversely, our bottom-up collapsing has similar behaviour at all levels of the tree. While the bottom-up approach results in a higher slot utilization on average, thus being more memory efficient, its behaviour is sub-optimal with respect to the SAH since a node with empty slots can improve the SAH by pulling up grand-children, which is what the top-down approach does. The bottom-up approach produces deeper trees with an average child area that decreases slowly compared to top-down collapsing (second row of Figure 4), which is caused both by the lower slot utilization at higher levels and the fact that top-down collapsing greedily minimizes child areas. The merge penalty does improve the rate of decrease of node areas for the BVH4, but has little impact on the BVH8. Finally we evaluate the impact of the merge penalty on the BVH quality and raytracing performance in Table 3, which shows that the merge penalty improves the trace time, SAH, and number of traversed nodes on average. As suggested by the results of Figure 4, it is much more effective for 4-wide hierarchies than 8-wide hierarchies.

Node ordering To measure the effectiveness of our partial reordering (section 3.4) we compare the raytracing performance of the same wide BVH without node reordering against partial breadth-first reordering and full breadth-first reordering. We use the performance of the non-reordered BVH as a baseline and show in Figure 5 the increase in raytracing throughput that we obtain using our partial reordering (in blue) compared to full reordering (in orange). Partial reordering results in a 4-19% increase in raytracing throughput for the BVH4 and 0-10% for the BVH8. Since this additional reordering step is very cheap (1% of build time on average), it can be enabled for all scenes as even when it doesn't increase tracing performance (on *Rungholt* for example), the additional cost is minimal. Since it is limited to the first levels of the hierarchy, partial reordering does not match the tracing performance of a fully reordered tree, showing a 33% difference on *Bistro Exterior* in the worst case. The BVH8 is overall less sensitive to node ordering than the BVH4 which is expected since spatial locality matters less when the size of the structure approaches the size of a cache line (128 bytes on our machine).

Conclusion We presented a novel approach for the construction of wide BVHs on the GPU without traversing a binary BVH. By fusing a bottom-up collapsing procedure with H-PLOC, we achieve significant reductions in build time compared to previous approaches. This makes our algorithm well-suited for dynamic
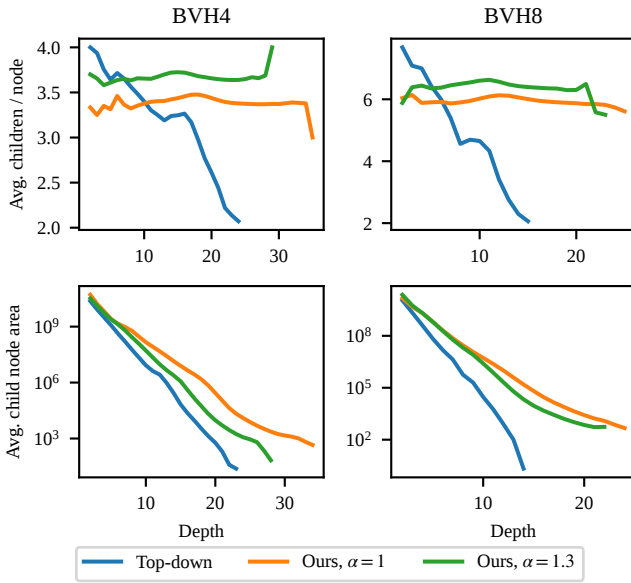
Figure 4: *We measure the average number of children per node, and average child area per level on the Powerplant scene. We compare the results between top-down collapsing (blue), our bottom-up collapsing without merge penalty (orange) and with merge penalty (green).*
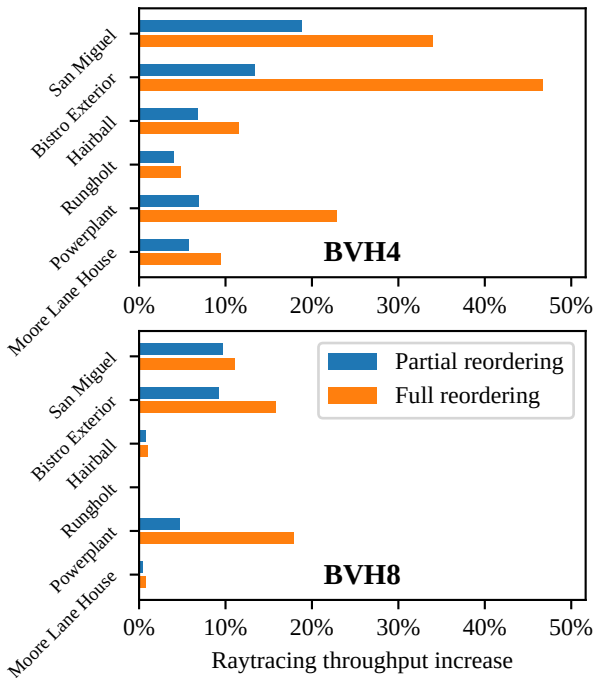

Figure 5: *Relative raytracing performance increase of node reordering, comparing the tracing throughput of a partially reordered BVH (section 3.4) and fully reordered BVH against a non-reordered BVH.*

scene rendering where BVH construction has a major impact on the total frame time. Our proposal can be implemented easily with a few modifications to an existing H-PLOC builder. Extending this method to a wider set of BVHs such as hierarchies with $K > 8$ or using a TLAS/BLAS decomposition would be a worthwhile avenue for future work. We are also interested in a deeper analysis of the impact of the merge penalty and node ordering to enable better tracing performance using improved heuristics.

References

[AM22] ADINETS A., MERRILL D.: Onesweep: A faster least significant digit radix sort for gpus. *arXiv preprint arXiv:2206.01784* (2022). 1

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)* (2014). 1

[BDTD22] BENTHIN C., DRABINSKI R., TESSARI L., DITTEBRANDT A.: Ploc++ parallel locally-ordered clustering for bounding volume hierarchy construction revisited. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 5*, 3 (2022), 1–13. 1

[BMB*24] BENTHIN C., MEISTER D., BARCZAK J., MEHALWAL R., TSAKOK J., KENSLER A.: H-ploc: Hierarchical parallel locally-ordered clustering for bounding volume hierarchy construction. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 7*, 3 (2024), 1–14. 1, 2, 3, 5

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *Computer Graphics Forum* (2008), vol. 27, Wiley Online Library, pp. 1225–1233. 2

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (2015), pp. 13–20. 1

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

[Gut14] GUTHE M.: Latency considerations of depth-first gpu ray tracing. In *Eurographics (Short Papers)* (2014), pp. 53–56. 2

[Hov] HOVHANNISYAN M.:. URL: https://mes-3d.gumroad.com/l/uukkwx. 9

[Int] INTEL:. URL: https://dpel.aswf.io/4004-moore-lane/. 9

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), pp. 89–99. 1

Wilhem Barbier & Mathias Paulin / Fused Collapsing for Wide BVH Construction

[Lai10] LAINE S.: Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics* (2010), Citeseer, pp. 107–111. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 375–384. 1

[LSS18] LIER A., STAMMINGER M., SELGRAD K.: Cpu-style simd ray traversal on gpus. In *Proceedings of the Conference on High-Performance Graphics* (2018), pp. 1–4. 2

[MB17] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE transactions on visualization and computer graphics 24*, 3 (2017), 1345–1353. 1, 3

[McG17] MCGUIRE M.: Computer graphics archive, July 2017. https://casual-effects.com/data. URL: https://casual-effects.com/data. 9

[MKVH24] MEISTER D., KULKARNI P., VASISHTA A., HARADA T.: Hiprt: A ray tracing framework in hip. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 7*, 3 (2024), 1–18. 2, 5

[MPS*22] MEINL F., PUTICA K., SIQUEIRA C., HEATH T., PRAZEN J., HERHOLZ S., CHERNIAK B., KAPLANYAN A.: Intel sample library, 2022. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html. 9

[Pin10] PINTO A. S.: Adaptive collapsing on bounding volume hierarchies for ray-tracing. In *Eurographics (Short Papers)* (2010), pp. 73–76. 2

[PL10] PANTALEONI J., LUEBKE D.: Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), pp. 87–95. 1

[SJ17] STEHLE E., JACOBSEN H.-A.: A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 417–432. 1

[TDDB23] TESSARI L., DITTEBRAND A., DOYLE M. J., BENTHIN C.: Stochastic subsets for bvh construction. In *Computer Graphics Forum* (2023), vol. 42, Wiley Online Library, pp. 255–267. 2

[VWB19] VAIDYANATHAN K., WOOP S., BENTHIN C.: Wide bvh traversal with a short stack. In *Proceedings of the Conference on High-Performance Graphics* (2019), pp. 15–19. 2

[Wal07] WALD I.: On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing* (2007), IEEE, pp. 33–40. 1

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

[YKL17] YLITIE H., KARRAS T., LAINE S.: Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics*. 2017, pp. 1–13. 1, 2, 4, 5

[YM] YOON S.-E., MANOCHA D.: URL: http://gamma.cs.unc.edu/DYNAMICB/. 9