H-PLOC
Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction

CARSTEN BENTHIN, Advanced Micro Devices, Inc., Germany
DANIEL MEISTER, Advanced Micro Devices, Inc., Japan
JOSHUA BARCZAK, Advanced Micro Devices, Inc., USA
ROHAN MEHALWAL, Advanced Micro Devices, Inc., USA
JOHN TSAKOK, Advanced Micro Devices, Inc., USA
ANDREW KENSLER, Advanced Micro Devices, Inc., USA

We propose a novel GPU-oriented approach for constructing binary bounding volume hierarchies (BVHs) based on the parallel locally-ordered clustering (*PLOC/PLOC++*) algorithm. Compared to competing high-performance GPU BVH build algorithms (*PLOC++* or *ATRBVH*), our method provides similar BVH quality in just a single kernel launch while achieving 1.1-3.6× lower construction times for the entire BVH build and 1.6-13× lower for just the binary BVH construction phase. Additionally, we propose an efficient algorithm to convert a binary BVH to an n-wide BVH with just a single kernel launch. Besides being extremely efficient, our proposed algorithms are simple to implement, allowing easy integration into existing frameworks.

CCS Concepts: • **Computing methodologies → Ray tracing**; • **Theory of computation → Sorting and searching**; **Massively parallel algorithms**.

Additional Key Words and Phrases: bounding volume hierarchy, ray tracing

ACM Reference Format:
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.* 1, 1, Article 1 (July 2024), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The bounding volume hierarchy (BVH) is one of the most popular acceleration structures in rendering used on both CPU and GPU. Its most common form in the context of ray tracing is a binary BVH, where each node has two children (i.e., a branching factor of two), and the bounding volumes are axis-aligned bounding boxes (AABBs). In the context of interactive/real-time workflows, reducing BVH construction times has become increasingly important as BVHs need to be either refit or rebuilt per frame to support dynamic content. BVH refitting is faster than a full rebuild but depending on primitive motion, it can cause severe BVH quality degeneration. BVH construction algorithms based on agglomerative clustering or treelet restructuring, which build the hierarchy in a bottom-up fashion, have become a favorable solution for contemporary GPU architectures, as their task distribution scheme is more suited to the GPU's massively parallel compute architecture than the traditional top-down approaches.

Authors' Contact Information: Carsten Benthin, Advanced Micro Devices, Inc., Germany; Daniel Meister, Advanced Micro Devices, Inc., Japan; Joshua Barczak, Advanced Micro Devices, Inc., USA; Rohan Mehalwal, Advanced Micro Devices, Inc., USA; John Tsakok, Advanced Micro Devices, Inc., USA; Andrew Kensler, Advanced Micro Devices, Inc., USA.

In this paper, we propose a novel BVH construction algorithm based on the parallel locally-ordered clustering approach (PLOC) [Meister and Bittner 2018a] and *PLOC++* [Benthin et al. 2022]. The latter is considered one of the fastest high-quality bottom-up binary BVH construction algorithms for GPUs available. The biggest weakness of *PLOC++* is its iterative nature, where a kernel launch per iteration is required to merge cluster candidates and thereby construct the inner nodes. The number of cluster candidates of the next iteration depends on the outcome of the current iteration, introducing a costly host-device synchronization dependency chain. The length of this chain is approximately proportional to the maximum depth of the tree. Our novel approach completely removes the host-device communication by constructing the binary BVH in a single kernel launch. Using the publicly available implementation of *PLOC++* [Benthin 2023] as a baseline, our approach achieves up to 4.4× faster binary BVH construction times, leading to a total BVH build time reduction (including all preprocessing and postprocessing phases) of up to 1.8×. Compared to agglomerative treelet restructuring, *ATRBVH* [Domingues and Pedrini 2015], another widely used treelet reconstruction-based BVH build algorithm, we achieve even higher BVH build time reductions: up to 13× faster binary BVH and up to 3.6× faster total BVH construction time. Compared to both competing GPU BVH build algorithms, our method is significantly simpler, making it an easy-to-implement approach for any kind of real-time high-quality BVH construction scenario.

In addition to efficient binary BVH construction, we propose a novel algorithm to address the second most costly BVH construction phase, i.e., the conversion of a binary BVH to a wide BVH (i.e., BVH with a branching factor higher than two). Wide BVHs are typically employed by modern ray tracing frameworks and required by various ray tracing hardware implementations. Similar to our binary BVH construction algorithm, our conversion phase needs just a single kernel launch.

2 Related Work

Various approaches for BVH construction, in particular in the context of ray tracing, have been proposed over the years. We discuss only the most relevant work here, and refer the reader to the survey by Meister et al. [2021] for more details. Most state-of-the-art BVH construction algorithms that focus on providing high-quality BVH for fast ray traversal performance minimize a cost function known as the *surface area heuristic* (SAH) [Goldsmith and Salmon 1987].

LBVH Lauterbach et al. [2009] proposed one of the earliest construction algorithms for GPUs known as LBVH. The algorithm ignores the SAH and instead is based on sorting triangles along a Morton curve. The key observation is that the Morton curve defines an implicit hierarchical scene partitioning constructed by spatial median splits that can be easily converted to an explicit BVH. Karras [2012] reformulated the algorithm such that the BVH topology is constructed in a single kernel launch, but another bottom-up pass is still needed to fit the node bounding boxes. Apetrei [2014] proposed a bottom-up algorithm which constructs the BVH topology and fits the bounding boxes simultaneously in a single kernel launch. Vinkler et al. [2017] proposed extended Morton codes which encode not only spatial location but also the size of a triangle, improving the BVH quality with little overhead. However, even with this extension, the quality of the resulting BVHs is inferior to approaches which explicitly optimize the cost function. The strength of LBVH is its speed, and thus, it is used in combination with more advanced algorithms such as *build-from-hierarchy* [Hunt et al. 2007] (i.e., using the LBVH partitioning to restrict the search space of the local optimizations) or optimization techniques (i.e., using LBVH to construct an initial BVH).

Agglomerative clustering Walter et al. [2008] proposed using agglomerative clustering to construct a BVH in a bottom-up fashion in contrast to traditionally used top-down approaches, building BVHs of higher quality at the cost of higher construction times. The nearest neighbor search is the bottleneck of this algorithm; the authors proposed to accelerate it by a first-stage KD-tree. Gu et al. [2013] proposed a BVH construction algorithm known as *approximative agglomerative clustering* (AAC), designed for multi-core CPUs. The algorithm starts from the root, recursively partitioning initial clusters based on the partitioning given by LBVH. Once the number of clusters is lower than a given threshold, the algorithm starts merging to decrease the number of clusters under the threshold. Then, the clusters are merged with the cluster from the sibling with regard to the LBVH partitioning. This process continues until the whole BVH is constructed. To accelerate the nearest neighbor search, the authors cache the distances in a distance matrix, updating the distances only for newly constructed clusters.

PLOC Meister and Bittner [2018a] proposed *parallel locally-ordered clustering* (PLOC), adapting agglomerative clustering for GPUs. The nearest neighbors are searched along a list of clusters ordered by a Morton curve. The list of clusters can be more easily maintained and updated in parallel, unlike a KD-tree or a distance matrix. The algorithm iteratively merges multiple cluster pairs in parallel, but each iteration consists of multiple kernel launches. Benthin et al. [2022] proposed a method to fuse all the kernel launches per iteration in a single one; this approach is known as *PLOC++*, and it is considered one of the fastest high-quality BVH builders to date. Viitanen et al. [2018] proposed a dedicated hardware implementation of the PLOC algorithm.

Treelet Restructuring Algorithms based on treelet reconstruction take an initial BVH and perform incremental topology updates to minimize the cost function. As the BVH is already constructed, the cost function can be optimized globally (unlike construction algorithms). Karras and Aila [2013] proposed a GPU-based algorithm, restructuring treelets of a fixed size in parallel. The algorithm starts from the leaves, proceeding up to the root. Once the subtree has enough nodes to form a treelet, the algorithm reconstructs the treelet in a brute-force manner. An advantage is that treelet restructuring can be efficiently implemented via warp (wave) intrinsics [Nickolls et al. 2008]. Agglomerative treelet restructuring, *ATRBVH* [Domingues and Pedrini 2015], replaced the brute-force algorithm with agglomerative clustering, achieving the same quality at a higher construction speed.

Wide BVH Conversion The aforementioned construction methods build binary BVHs (BVH2) which typically have to be converted to wider BVHs. A BVH2 can be converted to a wide BVH in a top-down fashion [Wald et al. 2008] by replacing children (e.g., selecting the one with the largest surface area) by grandchildren until all slots of a n-wide BVH node are occupied. This process continues recursively. Pinto [2010] and Ylitie et al. [2017] proposed an algorithm performing this conversion in an SAH-optimal way. The algorithm is based on dynamic programming consisting of two passes, one bottom-up and one top-down. Firstly, the bottom-up pass computes for each inner node the costs of representing the corresponding subtree using various numbers of trees. Then, the top-down pass based on the costs from the first pass reconstructs the optimal wide tree.

3 Hierarchical PLOC (H-PLOC)

In this section, we describe our novel construction algorithm called *hierarchical PLOC* (H-PLOC), followed by implementation details. In the following, we will refer to a binary BVH as *BVH2* and use the terms *work item* and *wave* as equivalents to the CUDA terms *thread* and *warp* [Nickolls et al. 2008]. The common size of a wave (CUDA warp) is 32 work items (CUDA threads).

3.1 Background

The *H-PLOC* algorithm combines the strengths of LBVH [Apetrei 2014] and PLOC++ [Benthin et al. 2022]. We briefly review the details of both algorithms in this section.

LBVH [Apetrei 2014] builds the BVH in a bottom-up fashion, proceeding from the leaves up to the root, using just a single kernel launch. The algorithm keeps, for each node, a range of Morton codes belonging to the corresponding subtree. The parent of a node lies either in a position preceding the first element in the range or a position succeeding the last element in the range, which can be efficiently determined based on the Morton codes. Starting from the leaves, following the paths through the binary hierarchy up to the root fully in parallel requires atomic synchronization to ensure that only one of the two threads arriving at each inner node is allowed to continue.

PLOC++ [Benthin et al. 2022] builds the BVH iteratively, merging multiple cluster pairs in parallel in each iteration. Each iteration consists of three phases: nearest neighbor search, merging, and compaction. The list of clusters, initialized with primitive bounding boxes, is sorted based on a Morton curve. In the first phase, each cluster in the list searches for its nearest neighbor from its location within search radius R in parallel, selecting the one that minimizes the surface area of the merged bounding boxes. A segment of clusters corresponding to the work group plus additional border clusters is loaded into shared local memory prior to the actual search to reduce access latency. Benthin et al. [2022] showed the nearest neighbor search can be simplified further by searching only to the right and propagating the result by an atomic operation to neighboring candidates. In the second phase, cluster pairs are merged if the nearest neighbors mutually correspond. The first cluster in the cluster pair is replaced by the merged cluster, and the second cluster is marked as invalid. In the last phase, invalid clusters are removed by a compaction operation. As the work groups do not need to communicate with each other, all three steps can be fused into a single kernel.

3.2 Algorithm Overview

Our approach builds a BVH in a single kernel launch in the bottom-up fashion in the same manner as Apetrei [2014] for *LBVH* and unlike *PLOC++*. Similar to other build-from-hierarchy approaches, the partitioning given by *LBVH* serves only as a guide for the final BVH construction. We are not interested in the actual *LBVH* tree but only track the ranges of Morton codes for currently processed nodes; the nodes themselves are not actually stored. Besides the Morton codes, we keep the list of cluster indices for each inner node. Initially, each leaf is considered as a cluster. In each *LBVH* inner node, we concatenate the cluster lists of both children. If the number of clusters exceeds a predefined merging threshold (a parameter of the algorithm), we invoke a variant of the *PLOC++*-based cluster merging algorithm [Benthin et al. 2022] to reduce the number of clusters until it is below the threshold (which may require several merging and compaction iterations). Reducing the number to just below the threshold instead of reducing it to a single root delays merging and increases the probability of putting larger clusters higher up in the tree. This generally improves BVH quality for scenes with highly varying primitive sizes. The essential steps of our approach are illustrated in Figure 1.

When we reach the root node of the *LBVH* tree, we perform cluster merging repeatedly until only a single cluster remains. This cluster becomes the root node of the output BVH. Note that the cluster indices can be stored efficiently in sub-arrays corresponding to the Morton code ranges since the number of clusters is bounded by the length of the range. The cluster merging phases consist of finding the nearest neighbors among the clusters in the list and creating new cluster nodes from merged pairs. Each newly created cluster replaces one of its children in the cluster list, while the other is invalidated, followed by a cluster list compaction.

We maximize efficiency by choosing a merging threshold of half the GPU's wave size, which means the length of two concatenated cluster lists will be between half and full wave size. This allows for a more efficient merging and compaction of cluster lists (see Section 3.3).

3.3 Implementation

The main loop of our approach is illustrated in Algorithm 1. Initially, we assign one work item per leaf and follow the *LBVH* hierarchy upwards. For the bottom-up traversal, we maintain an array of parent node IDs (initialized to an invalid state) which is atomically updated to ensure that only one of the two paths arriving at an inner node is allowed to continue. The update of parent node IDs can be efficiently implemented by an atomic exchange operation: only if the exchange returned a valid ID is the work item allowed to continue. As paths quickly terminate, wave utilization - the

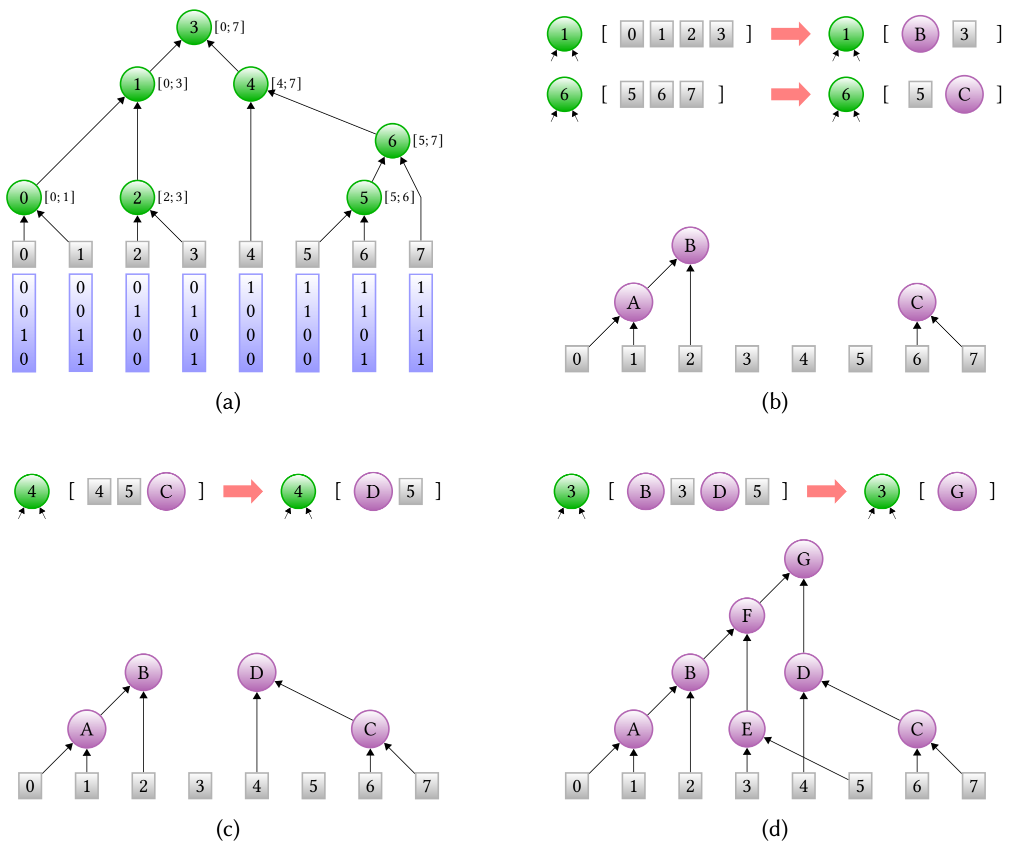


Fig. 1. Illustration of *H-PLOC*'s bottom-up construction process using a wave size of 4 and merging threshold of 2 (half the wave size): (a) Leaf nodes (grey) and inner nodes (green) of a binary hierarchy over 8 primitives defined by their Morton codes (light blue). Each (green) inner node covers a range of leaf nodes. The Morton code hierarchy only serves as a guide for constructing the final BVH2. (b) At inner nodes *1* and *6* (Morton code hierarchy), the number of clusters exceeds the merging threshold. *PLOC++*-based merging of clusters is executed, thereby constructing the inner nodes *A*, *B*, and *C* (violet) of the final BVH and reducing the number of clusters to equal or less than the threshold. (c) At inner node 4 of the Morton code hierarchy, again, the number of clusters exceeds the threshold. Merging is executed to build node *D*. (d) With inner node *3*, the root node has been reached, and the final BVH2 is completed with root node *G* via repeated merging.

Algorithm 1 Outer loop of *H-PLOC* algorithm: The Morton code-based bottom-up tree traversal is only used to combine neighboring ranges of cluster candidates. If after fusing two ranges, the number of candidates exceeds half the WAVE_SIZE, a *PLOC++*-based merging step is applied, which builds the final BVH2 and reduces the number of candidates below the threshold again. $findParentID$ returns the ID of the parent node (see Apetrei [2014]).

```
$B \leftarrow [B_0, .., B_{N-1}]$            ▷ BVH2 nodes, first $N$ entries are set to leaf bounding boxes
$MC \leftarrow [MC_0, .., MC_{N-1}]$         ▷ Morton codes sorted in ascending order
$I \leftarrow [I_0, .., I_{N-1}]$            ▷ cluster indices extracted from sorted Morton codes
$pID \leftarrow [pID_0, .., pID_{N-1}]$      ▷ BVH2 parent IDs, initialized to $-1$
for $i \leftarrow 0$ to $N-1$ in parallel do
    $L \leftarrow i, R \leftarrow i$                                ▷ Morton code range [L;R]
    $laneActive \leftarrow i < N$
    while $ballot(laneActive)$ do       ▷ Do bottom-up traversal as long as active lanes in wave
        if $laneActive$ then
            if $findParentID(L, R, N, MC) = R$ then
                $previousID \leftarrow atomicExchange(pID[R], L)$
                if $previousID \neq -1$ then
                    $split \leftarrow R + 1$
                    $R \leftarrow previousID$
                end if
            else
                $previousID \leftarrow atomicExchange(pID[L-1], R)$
                if $previousID \neq -1$ then
                    $split \leftarrow L$
                    $L \leftarrow previousID$
                end if
            end if
            if $previousID = -1$ then
                $laneActive \leftarrow false$
            end if
        end if
        $size \leftarrow R - L + 1$
        $final \leftarrow laneActive$ and $size = N$ ▷ Reached top of Morton tree, need to finish BVH2
        $waveMask \leftarrow ballot((laneActive$ and $(size > WAVESIZE/2)$ or $final)$
        while $waveMask$ do
            $laneID \leftarrow countTrailingZero(waveMask)$
            $plocMerge(laneID, L, R, split, final, I, B)$       ▷ Wave-based PLOC++ (Algorithm 2)
            $waveMask \leftarrow waveMask \,\&\, (waveMask - 1)$              ▷ Done with current lane
        end while
    end while
end for
```

number of active work items per wave - will drop accordingly. We restore high wave utilization for cluster merging and compaction by sequentially processing each active work item (path) of a given wave and its corresponding cluster list using the full wave through wave intrinsics (see Algorithm 2). As the merging threshold (the maximum length of a cluster list per inner *LBVH* node) is equal to half the wave size, the concatenated cluster lists will not exceed the wave size, and all merging and compaction iterations can be done within the wave. The cluster merging starts only when an inner *LBVH* node accumulates more than half the wave size clusters, hence only nodes higher up in the *LBVH* tree will trigger the process.

We store the list of cluster indices per LBVH node at the beginning of the corresponding Morton code range (see Algorithm 1), e.g., for an LBVH node which covers the range $[L; R]$ its left child covers range $[L; split - 1]$ while its right child covers $[split, R]$, and the cluster indices are stored at the beginning of respective ranges. Each cluster directly corresponds to a BVH2 node, and consists of a 3-dimensional AABB and two indices referring to its children (32 bytes total). For the nearest neighbor search in the cluster merging phase, we rely on a wave-based version of *PLOC++*'s search algorithm, which only requires *RS* cluster comparisons for a search radius of *RS*. The nearest neighbor search is the most inner-loop of the *H-PLOC* algorithm and, therefore, the most time critical. The number of clusters on which the nearest neighbor search operates varies between half- and full-wave size, and we measured an average wave utilization of 66%. We increase efficiency during nearest neighbor search further by caching all cluster list data in registers or shared local memory to reduce access latency and avoid global memory accesses. We allocate new clusters using a global atomic counter but reduce contention by sharing atomic increment requests per wave.

In terms of memory consumption for N primitives *H-PLOC* needs 64-bit Morton codes ($N \times 8$ bytes), parent IDs ($N \times 4$ bytes), cluster indices ($N \times 4$ bytes) and the BVH2 ($N \times 2 \times 32$ bytes). It is worth mentioning that most of these arrays can be placed into already allocated memory regions. For example, the BVH2 data can typically be placed into the memory region assigned for storing the primitive leaf data.

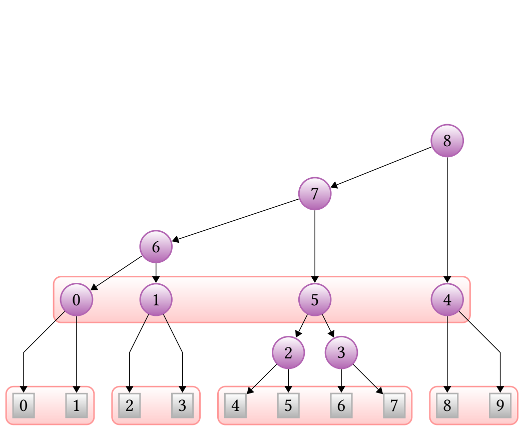
Algorithm 2 Inner loop of H-PLOC algorithm, entire wave performs nearest neighbor search and cluster merging for a single list of clusters to maximize parallelism. The maximum length of the cluster list is bound by the $WAVE\_SIZE$. *loadIndices* loads up to $WAVE\_SIZE$ elements from cluster indices I into per wave storage CI and *storeIndices* stores them into I again. $findNearestNeighbor$ and $mergeClustersCreateBVH2Node$ are wave-optimized versions of algorithms described in Benthin et al. [2022].

```
1:  function plocMerge($laneID, L, R, split, final, I, B$)
2:      $CI \leftarrow [CI_0, .., CI_{WAVE\_SIZE-1}]$                              ▷ per wave cluster indices
3:      $NN \leftarrow [NN_0, .., NN_{WAVE\_SIZE-1}]$                              ▷ per wave nearest neighbors
4:      $L_{start} \leftarrow waveBroadcast(L, laneID)$
5:      $R_{end} \leftarrow waveBroadcast(R, laneID)$
6:      $L_{end} \leftarrow waveBroadcast(split, laneID)$
7:      $R_{start} \leftarrow waveBroadcast(split, laneID)$
8:      $numLeft \leftarrow loadIndices(L_{start}, L_{end}, I, CI, 0)$             ▷ stored in $CI$ at offset 0
9:      $numRight \leftarrow loadIndices(R_{start}, R_{end}, I, CI, numLeft)$      ▷ stored in $CI$ at offset $numLeft$
10:     $numPrims \leftarrow numLeft + numRight$
11:     $THRESHOLD \leftarrow waveBroadcast(final) = true\ ?\ 1\ :\ WAVE\_SIZE/2$
12:     while $numPrims > THRESHOLD$ do
13:         $NN \leftarrow findNearestNeighbor(numPrims, CI, B, SEARCH\_RADIUS)$
14:         $numPrims \leftarrow mergeClustersCreateBVH2Node(numPrims, NN, CI, B)$
15:     end while
16:     $storeIndices(numPrims, CI, I, L_{start})$          ▷ store $numPrims$ indices into $I$ at offset $L_{start}$
17: end function
```

3.4 Conversion to Wide BVH

To achieve high performance, modern ray tracing frameworks and hardware implementations rely on wide BVHs. Nonetheless, the BVH construction algorithms almost exclusively produce binary BVHs that need to be converted to the wide format as a post-process. Surprisingly this conversion, which has non-negligible cost, has not been addressed much in previous work. We opt for a top-down traversal approach by Wald et al. [2008] that replaces the child with the largest surface area by its grandchildren, reducing the probability of overlapping child bounds. This can be implemented iteratively on the GPU with one kernel launch per level of the wide BVH. Nonetheless, similarly to *PLOC++*, this host-device synchronization dependency chain introduces unnecessary overhead. Hence, we propose an algorithm performing the conversion in a single kernel launch.

Our approach is based on maintaining an array of index pairs, where the first index references a binary BVH node and the second an index where the corresponding n-wide BVH node will be stored. Initially, the array contains a single pair referencing the root nodes of the respective binary and n-wide BVH. When the kernel is launched, each work-item is assigned a fixed position inside the index pair array, e.g., work-item 0 processes the first entry, work-item 1 the second, etc. The initial state of each slot is set to invalid, and each work-item continuously checks whether its array slot contains a valid entry. When a work-item picks up a valid entry, it traverses the binary BVH at the given index in a top-down manner (opening the child with the largest area) until n subtree indices are found. These n subtree indices are used to create an n-wide node at the given position in the n-wide BVH. The n subtree indices with n newly allocated n-wide BVH indices are re-inserted


(a)

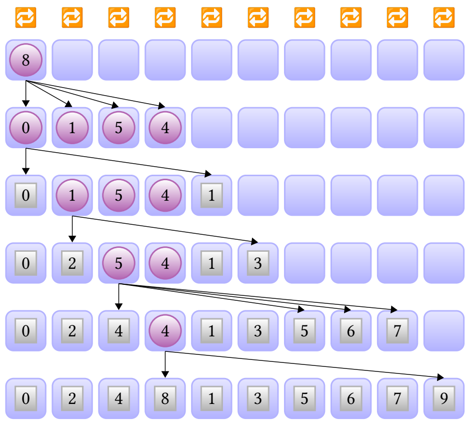

(b)

Fig. 2. Illustration of our fast top-down binary to n-wide BVH conversion algorithm, for the case $n = 4$. (a) BVH2 is traversed top-down, always opening up the child with the largest area. Opened-up children are used to fill the n-wide BVH node, in our example, a 4-wide BVH. The BVH2 nodes that correspond to the final 4-wide inner and leaf nodes are highlighted. (b) For efficient work distribution, we maintain an array of work assignments; each work-item is assigned a specific slot in this array, which continuously polls for work. The conversion starts with placing the BVH2 root node at slot 0. There, the first work item picks up the node and traverses the underlying tree in a top-down fashion, always opening up the child with the largest area. If $n = 4$ nodes have been accumulated, a 4-wide BVH node is created and the children are inserted in the work distribution array. The first child is inserted back into the slot of the current work-item; hence, $n - 1$ items are added (atomically) to the end of the current list. The process continues until the work distribution array contains only leaf nodes.

into the index pair array. The first entry is stored at the position assigned to the current work item, and the remaining *n-1* entries are appended after the last position in the index pair array. Note that the number of entries in the index pair array always grows, and no empty slot remains. The process continues until all entries in the array point to binary BVH leaf nodes. The number of binary BVH leaf nodes corresponds to the number of primitives the binary BVH is built over (one primitive per leaf), and we start the kernel with exactly this number of work-items. In order to guarantee forward progress, the work groups which make up the dispatch must start execution in order. If the underlying implementation does not guarantee this, it can be enforced by using an atomic increment to assign an increasing index to each work group as it is started. As each work-item reuses its original slot to store one of n newly added entries, it continues to make forward progress, potentially processing multiple entries until a binary BVH leaf node is reached, which cannot be opened further. The conversion algorithm is depicted in Figure 2.

Instead of always assigning one primitive to a n-wide BVH leaf node, SAH-based collapsing of binary BVH subtrees is possible with our conversion algorithm as well. Assuming SAH cost per inner binary BVH node is available, the algorithm can always decide to stop the opening process and emit a n-wide BVH leaf node, referencing multiple primitives.

4 Results

For our evaluation, we implemented *H-PLOC* in the HIP programming language (ROCm 5.7.3). All tests were conducted on an AMD® Radeon™ 7900 XT GPU and Ubuntu Linux 22.04. We compare *H-PLOC* against four other BVH build algorithms: *PRBVH* [Meister and Bittner 2018b], *ATRBVH* [Domingues and Pedrini 2015], *PLOC++* [Benthin et al. 2022], and *LBVH* [Apetrei 2014] in terms of BVH build times and quality using a collection of scenes with varying complexity (see Table 1). Extended Morton codes as proposed by Vinkler et al. [2017] have been used, although the impact on the SAH cost has mostly been negligible. For a better apples-to-apples comparison, we integrated the publicly available source code of each algorithm into a single code base, while paying careful attention that source code changes did not have any negative performance impact.

In general, a complete BVH build consists of multiple phases: setup, quadification (triangle-pairing), computing an AABB per primitive, Morton codes generation (based on AABBs) and sorting, fast binary BVH construction, and conversion to the final hardware-specific n-wide BVH format. Table 1 shows a breakdown of build times, BVH quality, and cost of different BVH build phases. Note that different BVH build algorithms only affect the binary BVH construction phase and, to some minor extent, the initial setup phase. All other phases share the same code. As our target ray tracing hardware is based on quads (triangle-pairs), we perform an initial quadification step, which does a parallel search for triangle pairs sharing a common edge. Triangle pairing reduces the number of primitives which are passed to the downstream phases by 20-50%. The actual BVH build time reduction depends on the build algorithm but is typically in a range of 10-60%. The final BVH format is a hardware-specific 4-wide BVH. Hence the conversion phase converts from a 2-wide to a 4-wide BVH using the algorithm described in Section 3.4. All BVH build algorithms use 64-bit Morton codes, combining the code and index into a 64-bit integer value, where 64-bit integer sorting is done by an optimized *OneSweep* radix sort [Adinets and Merrill 2022] implementation.

When comparing GPU BVH build algorithms, we first look at the two extremes: *LBVH* in terms of performance and *PRBVH* with respect to quality. *LBVH* just relies on the Morton codes to construct a binary BVH (bottom-up) in a single kernel launch. This is considered the speed-of-light in terms of binary BVH construction [Meister and Bittner 2022] thanks to the small amount of work the algorithm does, i.e., the child-parent node relation is already given by the Morton codes, and the BVH node bounding boxes just need to be propagated up the tree. Also, it does not need an atomic counter for inner node allocation, only for going up the Morton code tree. Due to relying on Morton

Table 1. Timings of BVH build phases (and relative percentage of total cost), total build time in ms, absolute build performance in mega-triangles per second and relative against *LBVH* (fastest build), and relative BVH4 SAH quality and path tracing run-times (*PT*) vs. *PRBVH* (highest quality). BVH build phases: setup and Morton code generation (*Misc*), finding triangle pairs (*Quad*), sorting of 64-bit Morton codes (*Sort*), construction of binary BVH (BVH2), and conversion to a *n*-wide BVH (*Conv*). *H-PLOC* provides similar BVH4 SAH quality and path tracing run-times (PT) as *PLOC++* or *ATRBVH*, while consistently outperforming *LBVH*. *H-PLOC* comes within 15% of *LBVH* build performance, outperforming *ATRBVH* by 2.7 − 3.6× and *PLOC++* by 1.1 − 1.8×.

	Misc	Quad	Sort	**BVH2**	Conv	Total	**Build Perf**	**BVH4**	
	[ms]	[ms]	[ms]	**[ms]**	[ms]	[ms]	**MTris/s**	SAH	**PT**
Sponza 0.3M Triangles									
PRBVH	0.10 (<0.1%)	0.19 (<0.1%)	0.09 (<0.1%)	**632** (99.9%)	0.19 (<0.1%)	633	**0.4 (<0.01)**	1.00	**1.00**
ATRBVH	0.09 (3.4%)	0.19 (7.2%)	0.09 (3.8%)	**2.09** (78.9%)	0.18 (6.8%)	2.65	**105 (0.25)**	1.10	**1.11**
PLOC++	0.12 (9%)	0.19 (15.0%)	0.09 (6.8%)	**0.71** (53.4%)	0.20 (15.8%)	1.33	**208 (0.49)**	1.11	**1.10**
LBVH	0.09 (13.6%)	0.19 (30.3%)	0.09 (13.6%)	**0.08** (12.1%)	0.19 (30.3%)	0.66	**422 (1.0)**	1.33	**1.28**
H-PLOC	0.10 (14.9%)	0.19 (25.7%)	0.09 (12.2%)	**0.16** (21.6%)	0.20 (25.7%)	0.74	**377 (0.89)**	1.09	**1.11**
Buddha 1M Triangles									
PRBVH	0.14 (<0.1%)	0.3 (<0.1%)	0.3 (<0.1%)	**2686** (99.9%)	0.39 (<0.1%)	2686	**0.4 (<0.01)**	1.00	**1.00**
ATRBVH	0.12 (2.6%)	0.3 (6.4%)	0.3 (6%)	**3.61** (76.8%)	0.39 (8.3%)	4.70	**231 (0.27)**	1.05	**1.04**
PLOC++	0.15 (6.9%)	0.3 (13.9%)	0.3 (13.9%)	**1.04** (48.1%)	0.37 (17.1%)	2.16	**503 (0.59)**	1.09	**1.06**
LBVH	0.12 (9.5%)	0.3 (23.4%)	0.3 (23.4%)	**0.2** (15.6%)	0.36 (28.1%)	1.28	**851 (1.0)**	1.14	**1.09**
H-PLOC	0.15 (10.1%)	0.3 (20.1%)	0.3 (20.1%)	**0.37** (25.5%)	0.37 (24.8%)	1.49	**728 (0.86)**	1.08	**1.05**
Hairball 2.9M Triangles									
PRBVH	0.20 (<0.1%)	0.6 (<0.1%)	0.26 (<0.1%)	**5808** (99.9%)	0.90 (<0.1%)	5810	**0.5 (<0.01)**	1.00	**1.00**
ATRBVH	0.18 (2.2%)	0.6 (7.3%)	0.26 (3.2%)	**6.18** (75.1%)	1.00 (12.3%)	8.23	**350 (0.28)**	1.04	**1.04**
PLOC++	0.21 (6.2%)	0.6 (18.2%)	0.26 (7.6%)	**1.44** (42.2%)	0.88 (25.8%)	3.41	**842 (0.69)**	1.06	**1.04**
LBVH	0.17 (7.2%)	0.6 (26.3%)	0.26 (11%)	**0.42** (17.8%)	0.89 (37.7%)	2.36	**1214 (1.0)**	1.10	**1.08**
H-PLOC	0.21 (7.9%)	0.6 (22.6%)	0.26 (9.8%)	**0.66** (24.9%)	0.90 (34.7%)	2.65	**1084 (0.89)**	1.05	**1.04**
Bistro 3.8M Triangles									
PRBVH	0.22 (<0.1%)	1.3 (<0.1%)	0.33 (<0.1%)	**12108** (99.9%)	1.41 (<0.1%)	12112	**0.3 (<0.01)**	1.00	**1.00**
ATRBVH	0.22 (1.5%)	1.3 (8.9%)	0.33 (2.3%)	**11.25** (77.6%)	1.41 (9.7%)	14.5	**267 (0.26)**	1.07	**1.08**
PLOC++	0.26 (4.6%)	1.3 (23%)	0.33 (6%)	**2.33** (41.5%)	1.37 (25%)	5.61	**691 (0.68)**	1.07	**1.07**
LBVH	0.22 (5.8%)	1.3 (34.3%)	0.32 (8.4%)	**0.62** (16.2%)	1.35 (35.3%)	3.82	**1013 (1.0)**	1.24	**1.28**
H-PLOC	0.26 (5.9%)	1.3 (29.9%)	0.33 (7.5%)	**1.13** (25.8%)	1.35 (30.8%)	4.38	**882 (0.87)**	1.07	**1.07**
San Miguel 10M Triangles									
PRBVH	0.74 (<0.1%)	2.3 (<0.1%)	0.7 (<0.1%)	**28297** (99.9%)	4.80 (<0.1%)	28306	**0.4 (<0.01)**	1.00	**1.00**
ATRBVH	0.67 (2.1%)	2.3 (7.1%)	0.7 (2.2%)	**23.94** (74.1%)	4.69 (14.5%)	32.30	**309 (0.30)**	1.10	**1.18**
PLOC++	0.80 (5.8%)	2.3 (16.6%)	0.7 (5.1%)	**5.31** (38.5%)	4.68 (34%)	13.78	**724 (0.71)**	1.10	**1.17**
LBVH	0.67 (6.8%)	2.3 (23.8%)	0.7 (7.1%)	**1.64** (16.6%)	4.55 (46.1%)	9.88	**1009 (1.0)**	1.35	**1.37**
H-PLOC	0.80 (7%)	2.3 (20.2%)	0.7 (6.1%)	**2.9** (25.3%)	4.74 (41.4%)	11.45	**871 (0.86)**	1.10	**1.17**
Powerplant 12.7M Triangles									
PRBVH	0.81 (<0.1%)	2.5 (<0.1%)	1.09 (<0.1%)	**38007** (99.9%)	5.59 (<0.1%)	38017	**0.3 (<0.01)**	1.00	**1.00**
ATRBVH	0.77 (2.2%)	2.5 (7%)	1.09 (2.2%)	**25.38** (72%)	5.54 (15.8%)	35.23	**362 (0.32)**	1.11	**1.10**
PLOC++	0.98 (6.8%)	2.5 (17.3%)	1.09 (7.6%)	**4.44** (31%)	5.32 (37.2%)	14.31	**892 (0.80)**	1.12	**1.11**
LBVH	0.76 (6.6%)	2.5 (21.8%)	1.09 (9.4%)	**1.86** (16.2%)	5.26 (45.9%)	11.46	**1112 (1.0)**	1.38	**1.30**
H-PLOC	0.98 (7.7%)	2.5 (19.7%)	1.09 (8.5%)	**2.75** (21.6%)	5.39 (42.4%)	12.71	**1004 (0.90)**	1.12	**1.13**

codes to define the binary BVH, the *LBVH* has the worst SAH quality, in particular for scenes with highly varying triangle sizes. All other build algorithms perform more work to improve the BVH quality, which increases build cost and, therefore, reduces BVH build performance. *LBVH* is used as the BVH build performance reference. *PRBVH*, based on parallel insertion, achieves the highest BVH quality (lowest SAH cost) and is used as a BVH quality baseline. *PRBVH*'s build times are generally too high for real-time usage and are more suitable for offline rendering.

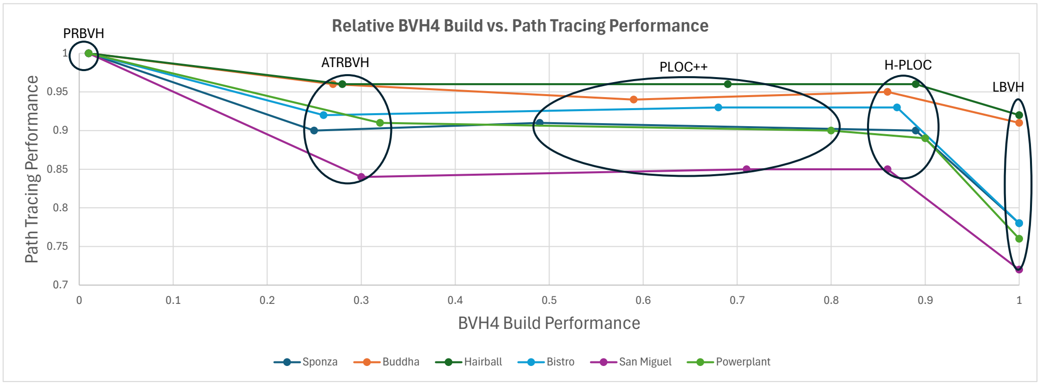


Fig. 3. Relative BVH build performance (against *LBVH*) vs path tracing performance (against *PRBVH*) for all BVH build algorithms and example scenes (data extracted from Table 1). *H-PLOC* provides similar path tracing performance than *ATRBVH* and *PLOC++*, while coming close to the BVH build performance of *LBVH*.

Besides the performance and quality extremes, we also compare against two other popular real-time GPU BVH build algorithms: *ATRBVH* and *PLOC++*. The quality settings of *ATRBVH* and *PLOC++* were adjusted to make them more comparable to *H-PLOC*, e.g., both *PLOC++* and *H-PLOC* use the same search radius of 8 for the nearest neighbor search. *ATRBVH*'s treelet size was set to 20, and the number of iterations was set to 1 to achieve high quality with a single kernel launch.

4.1 BVH Quality and Build Times Comparison

In terms of final BVH quality, *LBVH* has $1.10 - 1.38\times$ worse SAH cost than the *PRBVH* reference and thus the lowest quality overall. *ATRBVH*, *PLOC++*, and *H-PLOC* provide better quality and are only $1.04 - 1.12\times$ worse than *PRBVH*. Compared to each other, they provide roughly similar BVH quality across the different scenes, which also means *H-PLOC*'s merging of wave-sized cluster lists does not impose a reduction in the SAH cost compared to standard *PLOC++*, which iterates over far longer lists of clusters. In terms of the run-time performance impact the generated BVH has, we measured the time for tracing incoherent ray distributions generated by a path tracer. As BVHs constructed by our chosen algorithms exhibit quality differences for different parts of a given scene, we use multiple viewpoints and average the results. Compared to *PRBVH*, the relative run-time ray tracing overhead correlates mostly to the relative SAH cost, but depending on the scene and the selected views can differ slightly. Note that the SAH metric does not take into account memory access, caching effects, or traversal order which contribute to the difference to relative SAH cost.

While *PRBVH* provides the best BVH quality, it has at least two orders of magnitude higher build time than the other approaches. It is worth mentioning that *H-PLOC* does not seem to benefit from using a search radius larger than 8 as the SAH cost of the final 4-wide BVH basically stays flat with increased radius: for the given example scenes the maximum measured SAH cost variation was within 1-2%. A similar low SAH cost variation has been observed when using cluster lists larger than the GPU's wave size. Varying the cluster list size between 32 and 1024 work items showed a maximum SAH cost variation of 1.5%.

For the two competing approaches suitable for real-time, *ATRBVH* and *PLOC++*, the construction of the BVH2 and the conversion to an n-wide BVH are the most expensive phases. It is worth mentioning that *ATRBVH* uses many additional input buffers and costly distance matrix evaluations, which increase the algorithmic complexity and, therefore, run-time cost. The more efficient *PLOC++* has about 3-5× lower BVH2 construction times than *ATRBVH*. The build time for *PLOC++* is also

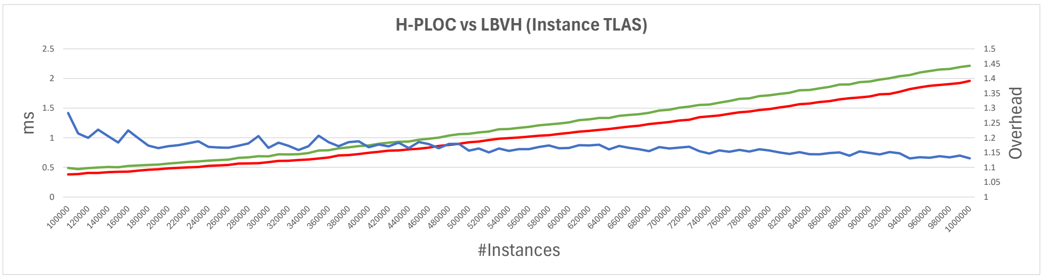


Fig. 4. *H-PLOC* (green) and *LBVH* (red) build times for building a 4-wide BVH over up to a million (random) instances (TLAS use case). With an increasing number of instances, the build time overhead of *H-PLOC* vs. *LBVH* (blue) quickly drops below 20% and settles around 15%. *H-PLOC* builds a high-quality 4-wide BVH over a million instances in just 2.21 ms vs 1.96 ms for a lower quality *LBVH*.

dominated by the BVH2 construction, which is 31-54% of the total time. The BVH2 cost for *H-PLOC*, is 1.6-4.4× lower than *PLOC++* and even 8.2-13× lower than *ATRBVH*, which brings the relative cost of BVH2 construction to below 26% of the total build time. This mostly shifts the bottleneck from BVH2 construction to the conversion phase. The BVH2 build time for *H-PLOC* is at most 2× higher than *LBVH*, and the overall build time is only about 15% higher, while the SAH quality is similar to *PLOC++* and *ATRBVH* (see Figure 3).

In terms of absolute build performance, *H-PLOC* achieves 0.8-1 billion triangles per second for the four largest models. The variation directly correlates to the average number of iterations of the *H-PLOC*'s inner loop (see Algorithm 2) to reduce the number of clusters to at most half the wave size. This iteration count varies between 1.8 for the faster builds (~1 GTriangles/s) and 2.3 for the slower builds (~0.8 GTriangles/s). The inner loop takes 50-60% of the total kernel run-time.

Today's ray tracing frameworks like DXR [Microsoft 2020] and Vulkan exhibit a two-level acceleration structure hierarchy: a single top-level (TLAS) built over multiple bottom-level acceleration structures (BLAS). Building a BVH4 over a triangle-based scene (as in Table 1) corresponds to building a BLAS, while a TLAS requires building a BVH4 over instances. An instance, in our case, requires 128 bytes (storing world-to-object transform and its inverse). Figure 4 shows *H-PLOC* and *LBVH* build times with an increasing number of randomly sized and placed instances. Similar to the BLAS case, the build time overhead of *H-PLOC* over *LBVH* for a larger number of instances quickly approaches 15%. In terms of absolute build times, *H-PLOC* builds a BVH4 over half a million instances in 1.1 ms and requires just 2.2 ms for a million instances.

4.2 Comparison to ATRBVH and AAC

One could observe certain similarities with *ATRBVH* [Domingues and Pedrini 2015] and *AAC* [Gu et al. 2013] since they both rely on the Morton curve and agglomerative clustering. In this section, we describe the major algorithmic differences.

ATRBVH is an optimization algorithm improving an existing BVH, typically constructed by *LBVH* but not necessarily limited to *LBVH*. In contrast, *H-PLOC* concurrently constructs the *LBVH* partitioning and uses it as a guide to build the final (binary) BVH in a single unified pass. Notably, the *LBVH* partitioning in *H-PLOC* is dynamically constructed on the fly, bypassing the need to write its nodes to global memory. Furthermore, *ATRBVH* engages in redundant efforts as restructured treelets exhibit substantial overlap. The algorithm assembles and restructures a treelet, ascends one level, and repeats the process, resulting in the top segment of the prior treelet being completely rebuilt, with only the lower part remaining unchanged. Another difference is the nearest neighbor search. *ATRBVH* caches distances between clusters in the distance matrix stored in the shared local memory, while *H-PLOC* searches in the cluster lists sorted along the Morton curve and then propagates the merged cluster lists up the hierarchy. Last, *ATRBVH* necessitates larger temporary storage than *H-PLOC* (refer to Section 3.3), accommodating distance matrices, SAH costs, and additional auxiliary buffers.

Although the computational steps of *H-PLOC* and *AAC* have a closer resemblance, *AAC* is tailored for multi-core CPUs, maintaining a substantial large computational state on the system memory stack, which would be infeasible for GPU builders. In contrast, *H-PLOC* is specifically designed for many-core GPUs, requiring only minimal state. Additionally, *AAC* requires a downward pass to find the *LBVH* partitioning while *H-PLOC* constructs the whole BVH in a single bottom-up pass. Similarly to *ATRBVH*, *AAC* caches the distances between clusters in a distance matrix.

5 Conclusion and Future Work

H-PLOC uses *LBVH* bottom-up construction as a guide and combines it with efficient *PLOC++*-based merging of wave-sized lists of clusters. This results in tremendous efficiency advantages, as *H-PLOC* outperforms competing binary BVH build algorithms by several factors while providing similar BVH quality. Considering the total BVH build cost, including all phases, the overhead of *H-PLOC* to the speed-of-light reference *LBVH* is only about 15%, while offering significantly better BVH quality. Combined with its implementation simplicity, which we believe is a strong argument for adoption, and the efficient binary to n-wide BVH conversion, *H-PLOC* could be the algorithm of choice for high-quality real-time BVH construction. In the future, we are interested in incorporating insertion-based refinement in the binary BVH construction phase to further lower the quality gap to *PRBVH*.

Acknowledgement

Model courtesy: San Miguel (Guillermo M. Leal Llaguno), Bistro (Amazon Lumberyard), Powerplant (University of North Carolina), Hairball (NVIDIA Research), Sponza (Crytek), Buddha (Stanford)

Copyright Notice and Trademarks

References

Andy Adinets and Duane Merrill. 2022. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs. (2022). https://arxiv.org/abs/2206.01784

Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. In *Proceedings of Computer Graphics and Visual Computing*.

Carsten Benthin. 2023. PLOC++ Implementation. https://github.com/embree/embree/tree/ploc

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 (2022).

Leonardo Domingues and Hélio Pedrini. 2015. Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring. In *Proceedings of High-Performance Graphics*. 13–20.

Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *Computer Graphics and Applications* 7, 5 (1987), 14–20.

Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH Construction via Approximate Agglomerative Clustering. In *Proceedings of High-Performance Graphics*. 81–88.

Warren Hunt, William Mark, and Don Fussell. 2007. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing*. 47–54.

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of High-Performance Graphics*. 33–37.

Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*. 89–100.

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.

Daniel Meister and Jiří Bittner. 2018a. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics* 24, 3 (2018), 1345–1353.

Daniel Meister and Jiří Bittner. 2018b. Parallel Reinsertion for Bounding Volume Hierarchy Optimization. *Computer Graphics Forum (Proceedings of Eurographics)* 37, 2 (2018), 463–473.

Daniel Meister and Jiri Bittner. 2022. Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing. *Journal of Computer Graphics Techniques* (2022).

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jirí Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum* (2021).

Microsoft. 2020. DirectX Raytracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (2008), 40–53.

André Susano Pinto. 2010. Adaptive Collapsing on Bounding Volume Hierarchies for Ray-Tracing. In *Proceedings of Eurographics (Short Papers)*.

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Aleksi Tervo, and Jarmo Takala. 2018. PLOCTree: A Fast, High-Quality Hardware BVH Builder. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018).

Marek Vinkler, Jiří Bittner, and Vlastimil Havran. 2017. Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction. In *Proceedings of High-Performance Graphics*.

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *Symposium on Interactive Ray Tracing*. 49–57.

Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. 2008. Fast Agglomerative Clustering for Rendering. In *Proceedings of Symposium on Interactive Ray Tracing*. 81–86.

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *High-Performance Graphics*. 4:1–4:13.