An evaluation of H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for BVH Construction

PATRICK ATTIMONT, Université Grenoble Alpes, France

Fig. 1. Visualization of BVHs constructed with H-PLOC in Nexus Renderer [3] for the Cannelle & Fromage scene, built in a few milliseconds. Color intensity represents the number of ray-node intersections (right).

Abstract. Bounding Volume Hierarchies (BVHs) are widely used in ray tracing applications to accelerate rendering by efficiently culling clusters of primitives. Is this project, we implement and evaluate H-PLOC [Benthin et al. 2024], a state-of-the art algorithm for constructing BVHs on GPUs. Building upon PLOC++ [Benthin et al. 2022], H-PLOC allows to build a BVH in a single kernel launch and achieves high efficiency by processing multiple clusters independently within each warp. We present a complete framework for fast BVH construction on GPUs using H-PLOC, evaluating its performance in terms of construction time and tree quality. Our results demonstrate that H-PLOC is able to rebuild BVHs for dynamic scenes with moving or animated objects in real-time. Additionally, we analyze the impact of key parameters of the algorithm on the final BVH structure.

1 Introduction

Bounding volume hierarchies (BVHs) represent an essential element of modern graphic applications using ray tracing. They significantly reduce the number of ray-primitive intersections required to determine visibility at a given pixel. Scene primitives (e.g., triangles) are grouped into hierarchical clusters, allowing large portions of the scene to be efficiently discarded for a given ray. A BVH in its binary form is a tree structure where each node represents an axis-aligned bounding box (AABB) enclosing a subset of the scene's primitives, and referring to both its left and right child nodes. The leaves of the tree contain one or more primitives, while the root node contains the entire scene. To evaluate the quality of such accelerations structures, Goldsmith and Salmon [1987] introduced the surface area heuristic: the probability of a node being intersected by a ray is approximately proportional to its surface area. A key objective in BVH construction is therefore to minimize the total surface area of the bounding volumes across the hierarchy, thereby maximizing the likelihood of discarding nodes during ray traversal. In other words, we aim to reduce the cost function defined as the sum of the surface area of all nodes in the tree.

However, increased quality often comes at the expense of construction time, which is a critical factor in real-time applications. As an example, ray tracing an animated object requires to rebuild the BVH at each frame to accommodate its changes in shape. In this context, the construction time

Author's Contact Information: Patrick Attimont, patrick.attimont@grenoble-inp.org, Université Grenoble Alpes, France.

of the BVH must be kept to a minimum to ensure real-time performance. Consequently, extensive research has focused on accelerating the construction time of BVHs, while maintaining a sufficient tree quality to ensure good tracing performance.

In this project, we evaluate the Hierarchical Parallel Locally-Ordered Clustering (H-PLOC) method [Benthin et al. 2024], a state-of-the-art parallel BVH construction algorithm that achieves significant results in both construction time and quality. We present an implementation of a GPU BVH builder in CUDA focused on building efficiency, and evaluate the performance of H-PLOC in terms of construction time and tree quality. We also provide an analysis of several parameters used by the algorithm and discuss their impact on the final BVH.

2 Background

The H-PLOC algorithm efficiently builds high-quality BVHs by combining two key concepts: Linear Bounding Volume Hierarchies (LBVH) and Parallel Locally-Ordered Clustering (PLOC). Traditional BVH construction methods rely on recursive top-down splitting, which becomes impractical for dynamic scenes. To address this, iterative and parallelized techniques, such as LBVH and PLOC, have been developed to accelerate construction while maintaining good tracing efficiency.

Iterative BVH Construction. First iterations of BVH construction algorithms were based on a top-down approach, where the tree is built recursively by splitting the scene into two parts at each level using spatial median splits. Goldsmith and Salmon [1987] later proposed the surface area heuristic (SAH) to evaluate the tracing efficiency of a BVH. Most state-of-the-art algorithms seek to minimize this cost function in order to build high-quality BVHs. For example, SBVH [Stich et al. 2009] uses SAH-based spatial splits and is considered the highest quality BVH builder. Although methods have been explored to improve build efficiency, iterative SAH builders are often slow and not well suited for dynamic or moving objects in real-time applications. Consequently, parallel BVH construction algorithms for both multi-core CPUs and GPUs have been developed to address this issue, the majority of which are based on a bottom-up approach.

LBVH. Lauterbach et al. [2009] introduced the concept of Linear Bounding Volume Hierarchies (LBVH) to allow for massively parallel construction of BVHs on GPUs. Morton codes (or keys) are used to sort primitives along a space-filling curve that preserves spatial locality. Each Morton code represents a primitive position in 3D space, and the curve ensures that nearby primitives have similar codes so that the sorting process will group them together. In an LBVH, each leaf node contains a single key, and each internal node corresponds to the longest common prefix in the subset of keys it represents. Apetrei [2014] improved the LBVH construction by introducing a new bottom-up traversal algorithm, where each nodes's parent can be retrieved in constant time. The layout of the tree is as follows: leaves are stored contiguously in memory, and internal nodes are stored in a separate array. As each leaf node contains one primitive and each internal node has only two children, the number of internal nodes is $n - 1$, for a primitive count of n. An internal node i will split the hierarchy between keys i and $i + 1$. The split point of an internal node corresponds to the last key of its left child, covering range $[l; i]$, and the first key of its right child, covering range $[i + 1; r]$. Consequently, the highest differing bit between the keys covered by an internal node will always be between keys i and $i + 1$. Such a layout allows to traverse the tree from the leaves up to the root: for any given node covering $[l; r]$, its parent is either at index $l - 1$ or r in the internal nodes array. Knowing which one is the parent is done by comparing the highest differing bit between keys $(l - 1, l)$ and $(r, r + 1)$. The one with the lowest differing bit position represents the parent, while the other corresponds to an ancestor. LBVH algorithms are considered the fastest methods to build BVHs, but they result in lower tree quality compared to SAH builders. An example of LBVH tree is given in figure 2.

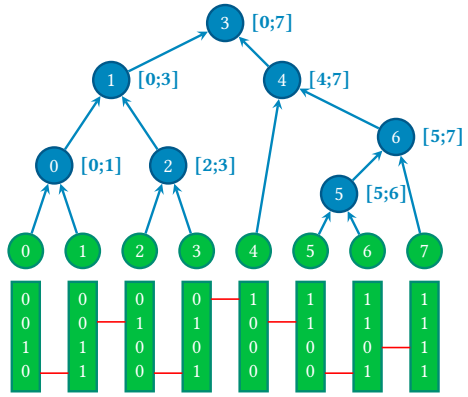

Fig. 2. An example of LBVH. Each leaf node (green), representing a key, is numbered from 0 to 7 and each internal node (blue) from 0 to 6. In a bottom-up construction, leaf nodes are processed first up to the root node, and the parent index of any given node is computed by comparing the highest differing bit at both its extremities (red lines). For example, node 2 covers keys 2 and 3 and its parent is at index 1 in the list of nodes since the highest differing bit between keys 1 and 2 is in a lower position than the highest differing bit between keys 3 and 4.

PLOC and PLOC++. Parallel locally-ordered clustering (PLOC) [Meister and Bittner 2018] is a bottom-up BVH construction method based on agglomerative clustering. In the same way as LBVH, primitives are ordered along a Morton curve and a cluster list is initialized with these sorted primitives. The algorithm iteratively merges multiple cluster pairs in parallel but requires three kernel launches per iterations. To address this issue, Benthin et al. [2022] introduced PLOC++, a variant of PLOC that reduces the number of kernel launches to one per iteration. At each iteration, three main steps are performed: nearest neighbor search, merging, and compaction. During the first phase, each cluster in the list searches for its nearest neighbor from its location within a search radius R. The distance metric between two clusters is the surface area of the AABB that bounds both clusters (the goal is to minimize the sum of the surfaces of all nodes in the tree). When two clusters mutually agree on being each other's nearest neighbor, they are merged and the first cluster in the pair is replaced by the newly created cluster, while the second cluster is marked as invalid. The last step consists of removing invalid clusters by a compaction operation.

3 H-PLOC Overview

H-PLOC builds upon PLOC++ and completely removes the need for launching several kernels, so that the entire tree construction is performed on the GPU and doesn't require any synchronization with the host. It uses LBVH partitioning as a guide for the tree construction, but does not actually store the LBVH tree. Each LBVH node is associated with a list of clusters in its range of Morton codes. Initially, each leaf (or primitive) is considered as a cluster. Cluster lists in two LBVH children are repeatedly concatenated as we go up the tree, until the number of clusters in an LBVH node exceeds a predefined threshold. A PLOC++ merging operation is then performed to reduce the number of clusters to just below the threshold. The newly created clusters correspond to new nodes in the constructed BVH. Once the root of the LBVH tree is reached, merging is performed repeatedly on the remaining clusters until only one cluster remains, which corresponds to the root of the BVH tree. The main steps of the algorithm are illustrated in figure 3.

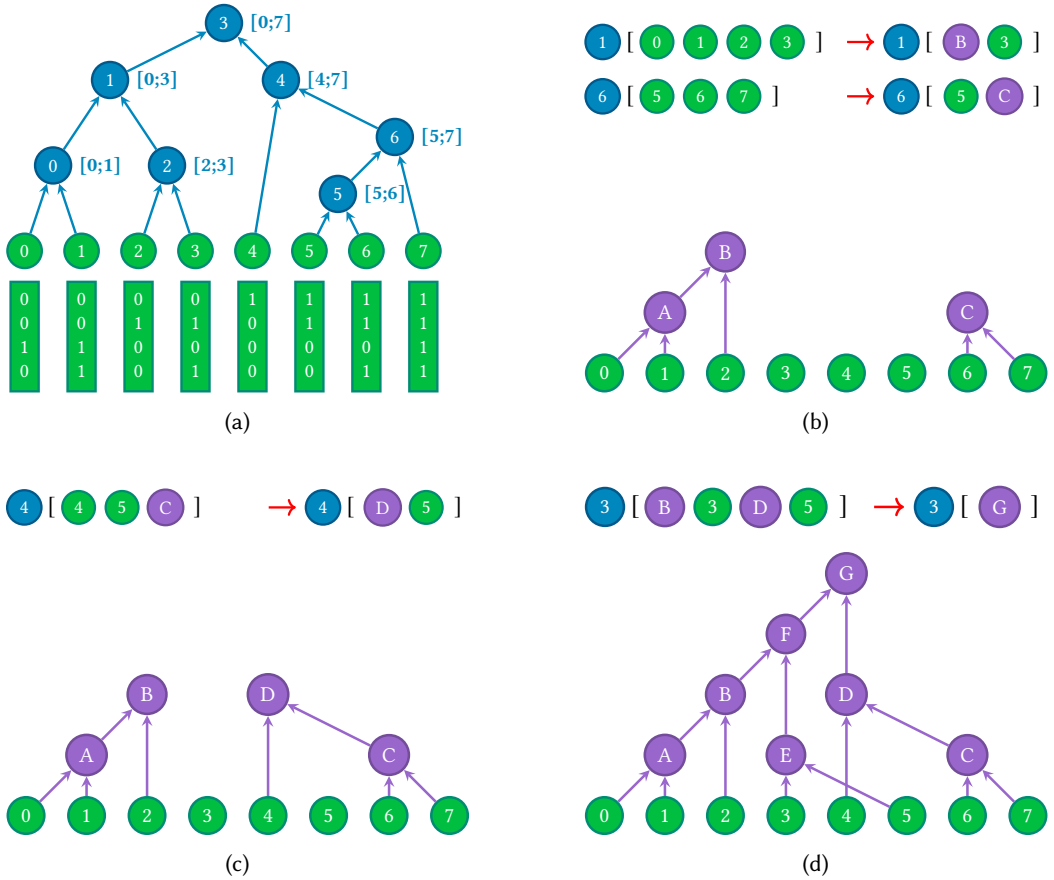

Fig. 3. An example of the H-PLOC construction algorithm as described by Benthin et al. [2024] with a merging threshold of 2. (a) Leaf nodes are first sorted according to their Morton codes and each lane start from the leaves and move up the LBVH tree at every iteration. (b) LBVH node 1 has 4 clusters (leaves from 0 to 4) which exceeds the merging threshold. Cluster A is created by merging leaves 0 and 1, then cluster B is created by merging cluster A and leaf 2. Cluster B and leaf 3 are the two remaining clusters in LBVH node 1. LBVH node 6 merges leaves 6 and 7 into cluster C. (c) Likewise, cluster D is created in LBVH node 4. (d) Once the LBVH root node is reached, remaining clusters are repetitively merged into one final cluster representing the root of the new BVH.

4 Implementation Details

We implemented the H-PLOC algorithm in C++ and CUDA as a BVH building library [4]. To test and evaluate our implementation, we integrated this library into a GPU-based ray tracing engine [3]. The implementation containing many specificities, we focus on detailing only the most significant aspects.

4.1 Bounding Boxes and Morton Codes Computation

The algorithm is first given a list of primitives, from which must be computed the bounding boxes and Morton codes. To compute the Morton codes, the 3D position of each primitive must be discretized within the bounding box of the entire scene. In a first pass, it is therefore necessary to

compute the bounding box of the scene in a separate kernel. To do this, we use a warp-wide reduction algorithm with `__shfl_down_sync` operation to perform parallel merging of the primitives bounding boxes in the scene. The reducing operation is a custom `atomicGrow` function computing the bounding box of two AABBs taken as input. Once each warp has computed its reduced bounding box, a scene bounding box stored in global memory is atomically updated. We also implemented a block-level reduction using shared memory, but found that for our use case with a few million primitives, the warp-level reduction was more efficient. In this first pass, the bounding boxes of primitives also serve to initialize the cluster list required for the H-PLOC algorithm.

The second step consists of computing the Morton codes of the primitives. Each primitive's centroid 3D position is discretized within the scene bounds along each axis. The discretization accuracy depends on the number of bits used to represent the Morton code. We implemented two versions of the Morton code computation kernel: one using 32-bit codes (10 bits per axis) and the other using 64-bit codes (21 bits per axis). The 32-bit version is faster but can lead to more clustering errors due to lower precision. The 64-bit version is slower but provides better results. The list of Morton codes is stored in global memory, along with the list of cluster indices initialized to the primitive indices.

Both the bounding box and Morton codes computation kernels use grid-stride loops (each thread processes multiple primitives) for better scalability and more flexibility in choosing launch parameters to improve occupancy.

The third pass consists of sorting the primitives according to their Morton codes. We use the Onesweep radix sort algorithm [Adinets and Merrill 2022] implemented in the CUB library to sort the Morton codes (keys) and cluster indices (values). The sorting step allows us to obtain an LBVH layout of the primitives, which is then used as a guide for the H-PLOC algorithm.

4.2 H-PLOC Implementation

To maximize efficiency, H-PLOC uses warp-level merging where each warp processes a cluster list, and each thread in a warp processes a cluster. Cluster indices and AABBs are shared within the warp using warp-level primitives, which completely removes the need for shared memory during nearest neighbor search and merging. The implementation is described below.

Main Iteration. Initially, each thread is assigned a leaf node and thus a single cluster and will move up the LBVH tree. A list of parent ids initialized to -1 is kept in global memory to allow only one of the two paths to reach the LBVH parent. To do this, both threads update the parent id with an `atomicExch` operation, and the thread with a valid parent id (different from -1) will continue to the next iteration. Consequently, work units quickly terminate as the number of active threads is divided by two on average at every iteration. To address this issue, H-PLOC makes inactive threads in a warp participate in the merging process of active ones. This is done by using a warp-wide ballot operation to determine which threads require merging (i.e., the length of their cluster list is greater than the merging threshold). Merging is then iteratively done by the entire warp for each of the active threads. H-PLOC sets a merging threshold of 16 so that two concatenated cluster lists will never exceed a length of 32, which is the size of a warp. Before the merging operation of an active thread, its cluster list is fetched from global memory to the warp (one cluster index and AABB per thread), and warp-level merging is performed. The list of cluster indices is stored per LBVH node at the beginning of their corresponding Morton code range. A cluster index directly corresponds to a node in the constructed BVH, which consists of an AABB and two indices referring to its left and right children.

Warp-Level Nearest Neighbor Search and Merging. The nearest neighbor search being the most inner loop of the H-PLOC algorithm, it represents the most time critical part of the kernel. We

use a warp-optimized version of the nearest neighbor search algorithm described by Benthin et al. [2022]. Given N clusters in a warp and a search radius of R, the search complexity is $N \times R$. Each thread processing cluster i explores up to R clusters to its right, updating simultaneously $\mathcal{D}_i(i+r)$ and $\mathcal{D}_{i+r}(i)$, where $\mathcal{D}_i$ represents the area distance function used in PLOC++ evaluated at thread cluster i. Consequently, at the end of the search, a thread's nearest neighbor has also been updated by its left neighbors within a radius R. We rely on warp-level primitive `__shfl_sync` to share cluster data (index, bounding box, and nearest neighbor) between threads. Eventually, the merging operation is performed for mutually agreeing neighbors: new BVH nodes (clusters) are created using a global node counter atomically updated by the first thread of each warp. The left cluster of a merged pair is replaced with the newly created cluster, while the right cluster is marked as invalid. A cluster compaction operation using `__fns` to count the number of remaining clusters is then performed.

4.3 BLAS and TLAS

As scenes often contain numerous objects, it is very common to use two types of BVH in a ray tracing application: a top-level acceleration structure (TLAS), and a bottom-level acceleration structure (BLAS). BLAS are BVHs built for a single object in the scene (often using triangles as primitives), while TLAS are BVHs built over the list of BLAS instances representing objects in the scene (using the BLAS bounding boxes as primitives). The TLAS is used to quickly update the scene when object transforms are changing (position, rotation, scale), and additionally allows to create multiple instances of the same BLAS in the scene with different transforms. To handle both types of BVH, we implemented a BVH builder that supports both triangles and AABBs as primitives. Both variants are essentially the same, the only difference being that the AABB version removes the need to compute the triangles bounding boxes in the first step.

5 Results

We evaluated our implementation on a AMD Ryzen 7 5700X equipped with a NVIDIA GeForce RTX 3070 GPU (8 Go). All tests use 32-bit Morton codes with a search radius of 8 and a merging threshold of 16, unless stated otherwise. Execution time of every kernel was measured from the host side using CUDA events and averaged over 100 iterations after a warmup of 50 iterations.

5.1 Performance

Timings for the different steps involved in the building process are reported in table 1. We used scenes of varying complexity, ranging from 0.3 million to 28.1 million triangles. In terms of raw

Scene (triangles)	Scene Bounds (ms)	Morton Codes (ms)	Radix Sort (ms)	BVH2 (ms)	**Total (ms)**
Sponza (0.3M)	0.06	0.04	0.22	0.37	**0.68**
Buddha (1.1M)	0.22	0.14	0.37	1.05	**1.78**
Hairball (2.9M)	0.55	0.31	0.89	2.10	**3.86**
Bistro (3.8M)	0.59	0.31	1.02	2.66	**4.58**
Powerplant (12.7M)	2.52	1.31	3.59	8.85	**16.27**
Lucy (28.1M)	5.78	3.07	7.98	22.20	**39.03**

Table 1. BVH construction times for different scenes. All times are in milliseconds and represent the kernel execution time measured from the CPU. Radix sort uses 32-bit Morton codes. BVH2 refers to the H-PLOC kernel with a search radius $R = 8$.

performance, our implementation is able to process 0.44 - 0.83 billion triangles per second with our testing setup and hardware.

Additionally, relative kernel run times are reported in figure 4. Timings remain consistent across different scenes, with the H-PLOC kernel taking between 55% and 60% of the total build time. When using 64-bit Morton codes, sorting times is approximately 3× longer and takes the most time of the entire build process. Other kernels are not significantly affected by the change in Morton code size.

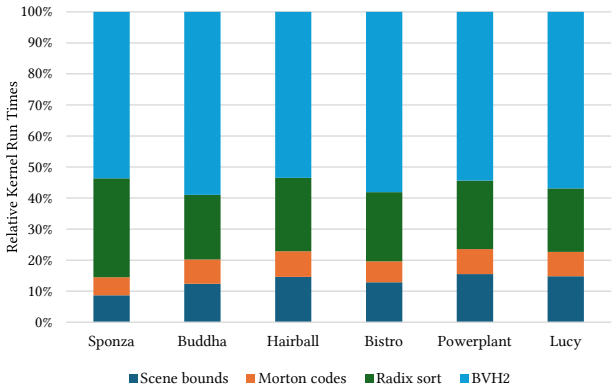

Fig. 4. Relative kernel run times measured from the CPU for different scenes. Timings remain consistent across different scenes.

Profiling. We used NVIDIA Nsight Compute to profile the different kernels. We studied the occupancy and throughput of the scene bounds and Morton code computation kernels. Both kernels are highly memory bound, which could be expected given their low arithmetic intensity. We found that the highest occupancy and performance was achieved with a block size of 64, although it might vary depending on the GPU model. Due to its independent warp-level processing of clusters, the H-PLOC kernel yields significant performance with an average of 64% in both compute and memory throughput. Storing cluster data in registers rather than in shared memory increases the register usage per thread to 60, but this remains acceptable and we found that build times are about 10% faster than when using shared memory.

5.2 BVH Quality

We compared the quality of the BVHs built using H-PLOC with LBVH for different scenes. An LBVH tree can be easily obtained with our implementation by using a merging threshold of 1 in the H-PLOC kernel, meaning two child clusters are always merged in the LBVH hierarchy. We implemented an evaluation kernel using the SAH metric defined by Meister et al. [2021], which gives the cost c of traversing root node N:

$$c(N) = \frac{1}{\mathcal{S}(N)} \left[c_t \sum_{N_i} \mathcal{S}(N_i) + c_i \sum_{N_l} \mathcal{S}(N_l) \right] \tag{1}$$

where $\mathcal{S}$ gives the surface area of a node, N_i and N_l denote the internal and leaf nodes of the tree, and constants c_t and c_i express the average cost of a traversal step and ray-primitive intersection, respectively. We used $c_i = 3$ and $c_t = 2$ in our tests. We report the cost of BVHs constructed with both methods for different scenes in table 2.

Scene (triangles)	Sponza (0.3M)	Buddha (1.1M)	Hairball (2.9M)	Bistro (3.8M)	Powerplant (12.7M)	Lucy (28.1M)
LBVH	296.8	211.3	1589.3	363.1	153.4	189.4
H-PLOC	216.6	183.0	1455.9	284.6	110.8	152.9

Table 2. SAH cost of BVHs constructed with H-PLOC and LBVH for different scenes.

We observe that BVHs built with H-PLOC have a significantly lower SAH cost than those built with LBVH, with a reduction of up to 28% in the Powerplant scene. This is due to the fact that H-PLOC is able to build more balanced trees by merging clusters in an SAH-optimized way.

5.3 Parameters Analysis

Search Radius. We evaluated the impact of the search radius R on the BVH cost and H-PLOC kernel time for the Bistro scene. The results are reported in figure 5. SAH cost decreases with increasing search radius, while the kernel time also increases. However, as noted by Benthin et al. [2024], the cost reduction is not significant for $R > 8$ (less than 2%). We choose $R = 8$ as it provides a good balance between construction time and tree quality.

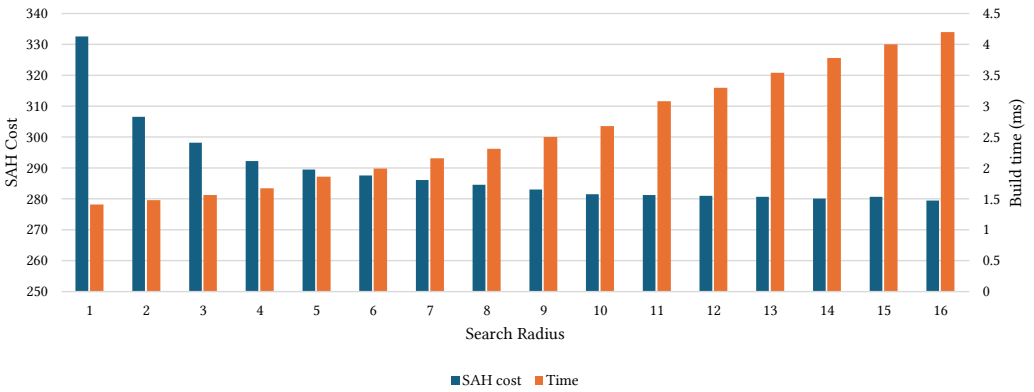

Fig. 5. BVH cost and H-PLOC kernel time for different search radii R in the Bistro scene. To obtain a good balance between construction time and tree quality, we choose $R = 8$.

Morton code length. To validate our implementation of 32-bit and 64-bit Morton codes, we used a scene composed of 20 million randomly generated triangles. By manually extending the scene bounds with different scale factors, we progressively lower precision in the discretization step used to compute the Morton codes. The BVH cost for both Morton code lengths is reported in figure 6. As expected, 32-bit Morton codes considerably increase SAH cost as scene bounds are extended.

Interestingly, we found that the BVH cost in our test scenes is not significantly improved by using 64-bit Morton codes, the most notable difference being a 0.8% reduction for Sponza scene. This is likely due to the fact that H-PLOC is able to find local neighbors in the merging process which seems to compensate for the limited precision in the discretized position. This is not the case, however, for LBVH whose structure is entirely determined by the Morton codes and does not rely on local clustering. Consequently, when using 64-bit Morton codes, we noticed an SAH cost reduction of up to 8.2% for the largest scenes in the LBVH structure.

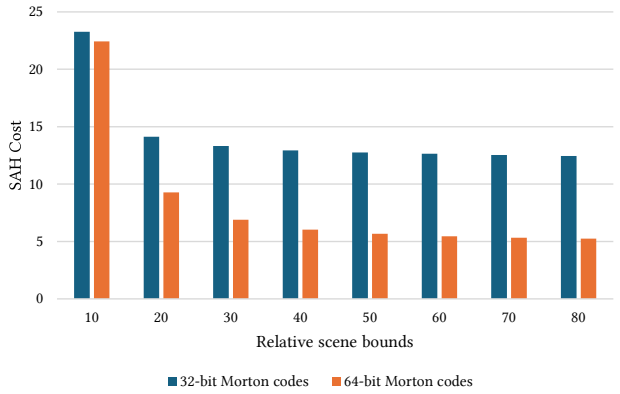

Fig. 6. SAH cost of BVHs built using 32-bit and 64-bit Morton codes for manually extended scene bounds with a scale factor of 10 up to 80. All tests use the same scene composed of 20 million randomly generated triangles.

6 Conclusion

In this project, we implemented and evaluated H-PLOC, a high-performance parallel BVH construction algorithm. Capable of processing up to 0.83 billion triangles per second, H-PLOC solves the challenge of rebuilding high-quality BVHs in real-time for dynamic scenes, even for meshes composed of millions of triangles. We also analyzed the impact of different parameters such as the search radius and Morton code length on the BVH cost and kernel execution time. Our implementation is available as a BVH building library and can be integrated into ray tracing engines. Although we focused on binary BVH construction, future improvements could include implementing the wide BVH conversion algorithm proposed by Benthin et al. [2024] to build BVHs with more than two children per node. Hierarchies such as compressed wide BVHs [Ylitie et al. 2017] have become increasingly popular due to their compact representation allowing faster ray tracing on GPUs.

Acknowledgements

Model courtesy: Bistro (Amazon Lumberyard), Powerplant (University of North Carolina), Hairball (NVIDIA Research), Sponza (Crytek), Buddha (Stanford), Lucy (Stanford), Dragon (Stanford), Cannelle & Fromage (LuxCoreRender).

References

Andy Adinets and Duane Merrill. 2022. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs. arXiv:2206.01784 [cs.DC] https://arxiv.org/abs/2206.01784

Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. The Eurographics Association. https://doi.org/10.2312/cgvc.20141206

Patrick Attimont. 2024. Nexus Renderer: an Interactive GPU Path Tracer Written in C++ and CUDA. https://github.com/StokastX/Nexus.

Patrick Attimont. 2025. Nexus BVH: A Fast and High Quality GPU BVH Builder. https://github.com/StokastX/NexusBVH.

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

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

Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications* 7, 5 (1987), 14–20. doi:10.1109/MCG.1987.276983

Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics (EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 33–37. doi:/10.2312/EGGH/HPG12/033-037

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

Daniel Meister and Jiří Bittner. 2018. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics* 24, 3 (2018), 1345–1353. doi:10.1109/TVCG.2017.2669983

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jirí Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. (2021). https://doi.org/10.1111/cgf.142662

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

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