PLOC++ : Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited

CARSTEN BENTHIN, Intel Corporation, Germany
RADOSLAW DRABINSKI, Intel Corporation, Poland
LORENZO TESSARI, Intel Corporation, Germany
ADDIS DITTEBRANDT, Intel Corporation, Germany

We propose a novel version of the GPU-oriented massively parallel locally-ordered clustering (*PLOC*) algorithm for constructing bounding volume hierarchies (BVHs). Our method focuses on removing the weaknesses of the original approach by simplifying and fusing different phases, while replacing most performance critical parts by novel and more efficient algorithms. This combination allows for outperforming the original approach by a factor of $1.9 - 2.3\times$.

CCS Concepts: • **Computing methodologies** → **Ray tracing**.

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

ACM Reference Format:
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. https://doi.org/10.1145/3543867

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. branching factor of two) and bounding volumes are axis-aligned bounding boxes (AABBs). In the context of interactive/real-time workflows reducing BVH construction times has become more and more important as BVHs need to be rebuild per frame to support dynamic content. In general, BVH construction algorithms for ray tracing can be categorized in top-down, bottom-up and insertion-based approaches. Bottom-up approaches have been very popular on GPU architectures, as their task distribution scheme is more suited to the GPU's massively parallel compute architecture. However, implementing an efficient and high quality bottom-up BVH builder on a given GPU architecture is still an issue [Meister et al. 2021].

In this paper, we propose a new version of the parallel locally-ordered clustering (*PLOC*) approach by Meister et al. [2018a], which is considered one of the fastest high-quality bottom-up BVH construction algorithms for GPUs. We show that by carefully simplifying and fusing different phases of the approach and replacing the time-critical ones with novel algorithms, we outperform prior work by a factor of $1.9 - 2.3\times$. Using the publicly available implementation [Meister 2018] as

Authors' addresses: Carsten Benthin, Intel Corporation, Germany, carsten.benthin@intel.com; Radoslaw Drabinski, Intel Corporation, Poland, radoslaw.drabinski@intel.com; Lorenzo Tessari, Intel Corporation, Germany, lorenzo.tessari@intel.com; Addis Dittebrandt, Intel Corporation, Germany, addis.dittebrandt@intel.com.

2577-6193/2022/7-ART31 $15.00
https://doi.org/10.1145/3543867

a baseline, our version does not only double performance, but also reduce memory requirements during construction.

In the following we will refer to Meister's original implementation as *PLOC* and to our new approach as *PLOC++*. With respect to the GPU programming model we will use the general OpenCL terms, e.g., an OpenCL work group corresponds to a CUDA block, an OpenCL computing unit to a CUDA streaming multiprocessor, an OpenCL work item to a CUDA thread, etc.

2 RELATED WORK

Various approaches for BVH construction, in particular in the context of ray tracing, have been proposed over the years. 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 *surface area heuristic* (SAH) [Aila et al. 2013; Goldsmith and Salmon 1987]. Recent top-down BVH construction approaches have seen improvements in construction time while maintaining high SAH quality, either by improved triangle splitting [Ganestam and Doggett 2016] or using an auxiliary LBVH [Lauterbach et al. 2009] for progressive tree refinement [Hendrich et al. 2017]. Further work on LBVH can be found in [Vinkler et al. 2017]. Bottom-up builders, instead, aggregate primitives starting from the leaves until the full tree is built; the required nearest neighbor lookup can be accelerated by kd-trees [Walter et al. 2008] or by using a Morton curve [Gu et al. 2013; Meister and Bittner 2018a]. Insertion-based methods work by adding primitives to an incremental data structure [Bittner et al. 2013], and have been recently parallelized on GPUs [Meister and Bittner 2018b]. For a detailed and comprehensive overview on top-down, bottom-up, and insertion-based approaches we refer to the latest state-of-the-art report on BVH construction algorithms [Meister et al. 2021]. Our work focuses exclusively on the parallel locally-ordered clustering approach for bottom-up BVH construction on GPUs [Meister and Bittner 2018a]. Besides the original GPU implementation, a dedicated hardware implementation has been proposed [Viitanen et al. 2018].

3 ORIGINAL PLOC

The original *PLOC* approach [Meister and Bittner 2018a] is an iterative algorithm using agglomerative clustering based on Morton curve-ordered [Lauterbach et al. 2009] cluster representatives. Cluster representatives are scene primitives' AABBs and inner BVH nodes. Besides an initial pre-processing step, each iteration consists of three phases: range-based approximate nearest neighbor search, locally ordered clustering to (implicitly) build a binary BVH, and a cluster representative compaction phase (see Figure 1). In the following we will illustrate the different phases of the original algorithm.

3.1 PLOC Pre-processing

The *PLOC* approach starts with a three step pre-processing phase: initialization of initial cluster representatives using AABBs of scene primitives, creation of centroid-based Morton codes from those and their sorting. The pre-processing phase is a prerequisite for the iterative BVH build process. Besides the cluster representatives, additional data is needed for the binary BVH nodes, the nearest neighbor information and the binary valid/invalid state per cluster representative, as well as auxiliary data buffers for prefix sum computation and compaction. Efficiently compacting cluster representatives in parallel requires double buffering, which is costly in terms of memory bandwidth and footprint. The publicly available implementation of *PLOC* avoids this bottleneck by introducing an additional indirection to the array of cluster representatives through using double-buffered index arrays (32-bit integer indices). All read/write accesses to the array of cluster representatives are done through the index arrays. Since both the initial cluster representatives (set to scene primitives' AABBs) and inner BVH nodes are considered cluster representatives during the build process, the

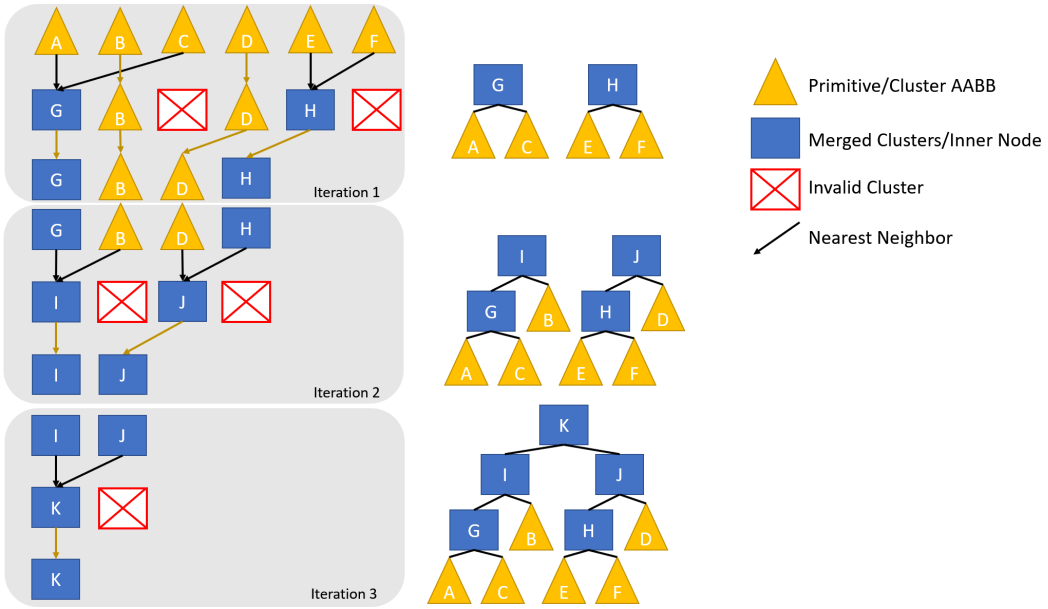

Fig. 1. Three *PLOC* iterations (left) for an array of six primitives. Each iteration consists of three phases: approximate nearest neighbor search among clusters, merging of cluster pairs, and cluster compaction (removing invalid clusters). The merging of clusters in each iteration creates an implicit binary BVH (right). The second child of the merged cluster pair is set to invalid and the compaction phase removes it before starting the next iteration. The algorithm terminates when only a single cluster representative remains.

former are simply treated as BVH leaf nodes, hence the integer indices point either to BVH leaves or inner nodes.

3.2 PLOC Iteration

Approximate Nearest Neighbor Search. Each *PLOC* iteration starts with a range-based approximate nearest neighbor search for each cluster representative. This search relies on the Morton-ordered cluster representatives, taking a predefined search radius into account. Given a search radius R and position i, the search scans the interval range $\langle i - R, i + R \rangle$ of representatives and determines the element that minimizes the chosen distance function (area of merged bounding boxes). The position of the element with lowest distance is saved as the nearest neighbor for position i. To improve the performance of the nearest neighbor search, the most expensive part of each iteration, a subset of cluster representatives are loaded from global memory into shared local memory first. This saves memory bandwidth and reduces access latency during the nearest neighbor search.

Locally Ordered Clustering. The second phase performs locally ordered clustering: If two representatives mutually agree on being their nearest neighbors ($neighbor[neighbor[i]] = i$), the pair will be merged and a new binary BVH node is created. The new BVH node automatically becomes a new cluster representative and it overrides one of the two input representatives in the index array, typically the one with the smaller index ($i < neighbor[i]$). The one with the larger index, instead, is no longer needed and is marked as invalid (using a separate valid/invalid state array).

Cluster Compaction. Invalid cluster representatives in the index array will be removed in the final compaction phase, which employs a prefix sum computation over all valid cluster representatives.

3.3 PLOC Bottlenecks

In terms of performance, Meister et al. [2018a] reports that *PLOC* is dominated by three factors: cost of nearest neighbor search, overhead of five kernel launches per iteration (one each for nearest neighbor search and merging and three for compaction) combined with a large number of iterations, and memory operations associated with auxiliary arrays (valid/invalid state, prefix sum, compaction). In the following we will propose a new version which addresses these shortcomings.

4 PLOC++

In the following we will illustrate our novel version of the PLOC algorithm. We start with the chunk-based nearest neighbor search (see Section 4.1), which allows for removing all auxiliary data buffers and merging the nearest neighbor search, the BVH construction phase and the prefix sum computation into a single kernel launch (see Section 4.2). Next, we reduce the complexity of the nearest neighbor search (see Section 4.3) itself, and present a straightforward path for efficiently handling the construction of the top of the binary BVH (see Section 4.4). Finally, we present a simple extension (see Section 4.5) which removes most of the kernel launch overhead altogether, while constructing a slightly different BVH.

4.1 Independent Chunk-based Approximate Nearest Neighbor Search

Our first modification to *PLOC* is to perform the nearest neighbor search for a chunk of cluster representatives locally within a work group and independently from other chunks. The nearest neighbor search for a given chunk can be performed independently if a small number of additional representatives of the neighboring left and right chunks (see Figure 2) are included in the search. Given a search radius of R, $2{\times}R$ representatives of the end of the left chunk and $2{\times}R$ of the beginning of the right chunk have to be included ($4 \times R$ total). Similar to *PLOC*, all required representatives for a given cluster, including additional border elements are loaded from global memory into shared local memory before the nearest neighbor search starts. The size of a chunk is therefore driven by the available shared local memory per work group (see Section 5). In general, it should be as large as possible to amortize the fetches of the additional border elements.

4.2 Merging Phases

The chunk-based nearest neighbor search (see Section 4.1) only requires cluster representatives as input. The output of the search for $chunk_i$ is a similarly sized array of nearest neighbor indices. These indices are all the data required to merge cluster representatives and create internal BVH nodes within the given chunk. It is important to note that $chunk_i$ can only merge representative X with its nearest neighbor if three conditions hold (see Figure 2): $neighbor[neighbor[X]] = X$, $X < neighbor[X]$ and $X \in chunk_i$. These rules allow for combining the nearest neighbor search and the merging of cluster representatives/BVH construction phase into a single *first* phase, which requires only a single kernel launch. As the nearest neighbor indices depend only on the chunk of representatives, they do not have to be stored in global memory anymore but instead can be kept in shared local memory.

On a successful merge, the index of the new representative (based on the new inner BVH node) overrides the previous one with the smallest index, while the other is marked as invalid. This update and invalidation output is written to the second array of representative indices. Non-merged representatives are simply copied from the first to the second array without any modification. The final compaction phase consumes the second array as input and writes the compacted sequence (removing invalid entries) into the first array of indices.

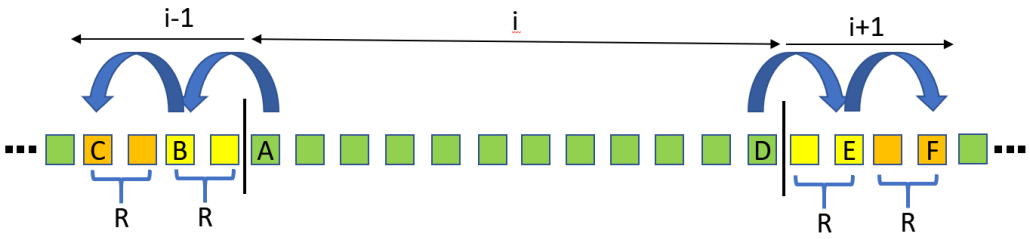

Fig. 2. Given a search radius R (in this example set to 2), the nearest neighbor search of a $chunk_i$ of N cluster representatives (green middle boxes) can be processed independently when including $2 \times R$ additional representatives on the left and right side of the chunk ($4 \times R$ total, orange and yellow boxes), which are small parts of the end/beginning of the neighboring left/right chunk i_{-1} and i_{+1}: if A (green) determined B (yellow) as nearest neighbor, we need to make sure that the nearest neighbor of B is A as well to successfully merge them. This requires scanning R additional elements to the left and right of B, e.g., element C (orange). It is important to note that cluster representative X will only be merged with $neighbor[X]$, if $neighbor[neighbor[X]] = X$, $X < neighbor[X]$ and $X \in chunk_i$ (green middle boxes). The latter ensures merging consistency between neighboring chunks, as only a $chunk_i$ is allowed to merge the pair. In the example, assuming $neighbor[A] = B \wedge neighbor[B] = A$, $chunk_i$ will *NOT* merge the pair but $chunk_{i-1}$ will as B has the smaller index and $B \in chunk_{i-1}$. Assuming $neighbor[D] = E \wedge neighbor[E] = D$, $chunk_i$ will merge this pair because D has the smaller index and $D \in chunk_i$.

Most of the prefix sum computation required for the compaction phase can be done within the *first* phase as well. Let C be the number of chunks and P be the minimal number of work groups which are needed to fully utilize the GPU. Chunks are now equally distributed over work groups, such that each gets a single range of C/P chunks. At the end of the *first* phase, each work group computes the number of valid cluster representatives within its assigned range of chunks, and saves the data into a small auxiliary array in global memory. This array needs only P entries and it can, for example, be held in the radix sort's histogram buffer. After the *first* phase is done, the compaction phase uses the same range of C/P chunks per work group assignment and computes a prefix sum over just P entries at startup. This fixed mapping of chunks to work groups avoids a separate kernel launch and the associated memory buffers to perform a global prefix sum computation as in the original *PLOC* approach.

The merging of phases in *PLOC++* brings the number of kernel launches per iteration down to two, which is significantly less compared to the five kernel launches of *PLOC*. Also, no additional global memory is required for storing nearest neighbors, prefix sums, and cluster representatives' valid/invalid state.

4.3 Reducing Nearest Neighbor Search Complexity

Of the two remaining kernels (see Section 4.2), the combined nearest neighbor search and BVH construction kernel is 10× more costly than the simple compaction kernel. The nearest neighbor search step itself is the dominating factor, i.e. given N cluster representatives and a search radius of R, the search complexity is $N \times R \times 2$. In the following we will show how to reduce the complexity to $N \times R$, cutting the cost in half.

Our algorithm relies on the commutative property of the distance function - surface area of the merged bounding boxes - with $distance(C_i, C_j) = distance(C_j, C_i)$. That means $distance(C_i, C_j)$ contributes to the nearest neighbor search for C_i and C_j. However, efficiently exploiting this commutative property is not straightforward as the distance computations happen in parallel and influence each others' nearest neighbor search.

Algorithm 1 Our new nearest neighbor search: first the nearest neighbor array N_i is initialized to a default value; next, for each cluster representative at position i, *SEARCH_RADIUS* elements are scanned to the right and $distance(C_i, C_{i+r})$ is computed for cluster C_i and C_{i+r}. The distance in 32-bit single precision floating point format is now treated as an unsigned integer, left-shifted by one (distance always positive) and the lowest bits are masked out to store the relative offset to the nearest neighbor. This final value is now used to atomically update the nearest neighbor entries at N_i and N_{i+r}. For a search radius of 16 only the four lowest bits of floating point mantissa are lost.

```
function encodeRelativeOffset(ID, neighbor)
    offset ← neighbor − ID
    return (abs(offset) ≪ 1) ∨ (offset ≫ 31)
end function
C ← [C_0, .., C_{n−1}]
N ← [N_0, .., N_{n−1}]
SEARCH_RADIUS_SHIFT ← 4                          ▷ search radius of 16 = 2^4
SEARCH_RADIUS ← 1 ≪ SEARCH_RADIUS_SHIFT
ENCODE_MASK ← ¬(1 ≪ (SEARCH_RADIUS_SHIFT + 1) − 1)
for i ← 0 to n − 1 in parallel do
    N_i ← MAX_UINT
end for
Barrier()
for i ← 0 to n − 1 in parallel do
    min_area_index ← MAX_UINT
    for r ← 1 to SEARCH_RADIUS do
        area ← distance(C_i, C_{i+r})
        area_i ← (bitcast<uint>(area) ≪ 1) ∧ ENCODE_MASK
        encode0 ← area_i ∨ encodeRelativeOffset(i, i + r)
        encode1 ← area_i ∨ encodeRelativeOffset(i + r, i)
        min_area_index ← min(min_area_index, encode0)
        atomic_min(N_{i+r}, encode1)
    end for
    atomic_min(N_i, min_area_index)
end for
```

We address this issue by first initializing the nearest neighbor index array per chunk (stored in shared local memory) to an invalid default state (maximum unsigned integer value). Next, for each cluster representative C_j we iterate over the interval $\langle j+1, j+R\rangle$ and compute the $distance(C_j, C_k), k \in \langle j+1, j+R\rangle$, which is then interpreted as an unsigned integer value (distance is always positive), left-shifted by one and the lowest bits are masked out to encode the relative position of its nearest neighbor (see Algorithm 1). This unsigned integer value is now used to atomically update the nearest neighbor entry at position j and k. The update is done by using a 32-bit atomic *min* operation. If the underlying GPU architecture supports efficient 64-bit atomics to shared local memory, the relative encoding is not required, as the 32-bit distance value and the 32-bit index can be encoded into the upper and lower parts of a 64-bit unsigned integer.

4.4 Efficient Construction of Upper BVH Levels

At the top of the BVH tree only a few cluster representatives remain active, the GPU becomes therefore heavily underutilized and the kernel launch overhead becomes even more relevant as the actual computational cost becomes smaller and smaller. A way of avoiding these kernel launches is to combine the two remaining kernels from Section 4.2 into a single kernel by restricting the execution to a single work group. The single work group allows for using work group barriers to synchronize between the merged nearest neighbor / BVH construction and compaction phases. This makes it now possible to perform the main iteration loop inside the kernel using work group barriers for synchronization. This removes the previous kernel launch overhead for the remaining iterations completely. On the host side, the switch to the single work group kernel is done if the number of active cluster representatives is less than a predefined threshold. For our implementation (see Section 5) a switch threshold of $4 - 8K$ primitives worked well for all tested example scenes.

4.5 Two-Level Hierarchy Extraction (*PLOC++ Two-Level*)

The same idea of letting *PLOC++* iterate locally within a work group using work group barriers for synchronization (see Section 4.4) can be applied to remove most of the kernel launch overhead for the entire iterative process. By identifying independent ranges of cluster representatives and letting each range be processed by a single work group, only a single kernel launch is required to start enough work groups to process all independent ranges. After this kernel launch, each range has its BVH built and a top-level BVH over these range BVHs needs to be built. The top-level BVH construction needs a second kernel launch. This two-level approach, which we call *PLOC++ Two-Level*, is similar to other implicit two-level hierarchy extraction and BVH construction approaches [Meister et al. 2021]. It is important to note that the implicit two-level approach generates a different BVH compared to the full-iterative version of *PLOC/PLOC++*, as the nearest neighbor search is limited to the cluster representatives within the independent range (see Section 5).

The partitioning of cluster representatives into independent ranges can be efficiently generated by using existing sorted Morton codes which define an implicit binary tree. Fully utilizing the GPU requires the number of independent ranges to be equal or greater than the minimal number of work groups required to utilize all computing units. In order to quickly extract a sufficient number of independent ranges, the (implicit) Morton binary tree is traversed top-down in a breadth-first manner. The root node of the Morton binary tree is associated with the full range over all cluster representatives, and the two children of each node split the given range into two independent sub-ranges. This splitting process tries to split the largest ranges first and continues until the total number of ranges reaches a predefined threshold.

Letting a work group build the BVH per range up to the root node is sub-optimal in terms of BVH quality as there can be large overlap between the root nodes' bounding boxes, leading to a lower quality top-level BVH. A better approach is to stop the iterative process earlier, i.e. if the number of active cluster representatives has reached a given threshold. This gives the top-level BVH build a better chance to reduce overlap, thereby increasing BVH quality. We have determined a stopping threshold similar to the nearest neighbor search radius to provide the biggest gain in top-level BVH quality (see Section 5.2).

4.6 Memory Consumption

The merging of different phases into a single kernel launch (see Section 4.2) made the explicit arrays for storing valid/invalid state per cluster representative and prefix sum data obsolete. For N scene primitives, *PLOC++* therefore needs $N \times 32$ bytes for the initial cluster representatives, $N \times 32$ bytes for the binary BVH nodes, and $2 \times N \times 4$ bytes for integer index arrays referencing

cluster representatives, which yields a total of $N \times 72$ bytes. The temporary memory for storing and sorting Morton codes is aliased with currently unused buffers, for example the BVH node memory which is not needed during the pre-processing phase. *PLOC++*'s reduced prefix sum computation and *PLOC++ Two-Level*'s independent ranges need a small additional amount of memory of approximately $P \times 2 \times 4$ bytes, where P is the minimal number of work groups needed to fully utilize the GPU. This small amount of memory is aliased with unused buffers as well.

4.7 Faster Conversions in Corner Cases

In terms of efficiency, *PLOC*'s original version of nearest neighbor search breaks down if the cluster representatives' bounding boxes are exactly the same. This corner case of lots of bounding boxes being located at the exact same position and having the same size occasionally happens in real-world content, in particular in scenes where many point primitives (particles) start at the same initial position. In this case, the nearest neighbor search for position i will determine the first element tested, which will always be at position $i-1$, if the scan starts to the left, as the nearest neighbor. Subsequently scanned elements won't update the position as the distance function won't produce a smaller result, hence the condition $neighbor[neighbor[X]] = X$ won't be fulfilled for any element except for the very first, where no element to the left exists ($neighbor[0] = 1, neighbor[1] = 0$). This causes the nearest neighbor search to only find a single pair per iteration, resulting in linear complexity in the number of equal bounding boxes.

We address this issue by initializing the local nearest neighbor array (see Section 4.3) for position i to $i+1$ if i is even and to $i-1$ if i is odd. Neighboring elements with equal bounding boxes will therefore be directly merged, resulting in logarithmic build complexity for the corner case.

5 RESULTS AND DISCUSSION

For our evaluation we implemented both *PLOC* and our *PLOC++* algorithm in oneAPI's DPC++ language [Intel Corporation 2021] and run on a pre-production Intel® Alchemist-G10 GPU [Intel Corporation 2022] (32 Xe cores, 16GB GDDR6 memory, 256-bit memory interface)[1] using Ubuntu 20.04 Linux. The maximum work items in a work group is 1024, and each work group can use up to 64K of shared local memory, while the nearest neighbor search and hierarchy construction phase only requires approximately 32K. Fully saturating the 32 Xe cores requires 64 work groups with 1024 work items each. Both implementations have been extensively optimized with respect to the underlying GPU architecture. 64-bit Morton codes (64-bit keys + 32-bit index = 96-bit element size) are used in both approaches as they provide higher quality for the nearest neighbor search in the more complex scenes. The Morton codes are sorted by an optimized LSB-based radix sort which requires eight iterations (8-bit radix) to sort the 64-bit keys. Instead of performing eight sorting passes on the full 96-bit sorting element, we split the sorting into two blocks of four passes, where only 32-bit keys are used. This reduces the sorting element size to 64-bit. We initialize the 32-bit key of the 64-bit element with the lower 32 bits of the 64-bit key. Before the second block, we replace the 32-bit key with the upper 32-bits of the 64-bit key. This 2×32-bit key sorting approach allows for keeping the element size during sorting iterations to 64-bit (8 bytes) instead of 96-bit (12 bytes), which saves memory and cache bandwidth and therefore reduces sorting time by up to 20%. For BVH construction speed and ray tracing performance comparisons, we used a selection of the freely available models from Morgan McGuire's graphics archive [McGuire 2017]. In the following we will focus on presenting relative BVH build performance against the original *PLOC* algorithm running on the same hardware platform. For absolute performance numbers see Appendix A.

[1]Performance varies by use, configuration and other factors. Learn more on the https://edc.intel.com/content/www/us/en/products/performance/benchmarks/overview/ Performance Index site.

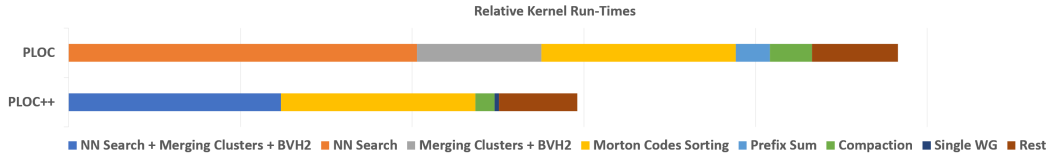

Fig. 3. Relative kernel run-times measured on the device (excluding kernel launch overhead). *PLOC++*'s fused nearest neighbor search - merge - BVH construction kernel (blue) requires just approximately half the time of *PLOC*'s previous two corresponding kernels (orange, grey). *PLOC++*'s single work group kernel (dark blue), constructing the upper levels of the BVH, and the simplified compaction kernel require just a fraction of the total run-time. The iterative kernels have now similar costs as the pre-processing kernels (*Sorting+Rest*).

5.1 BVH Build Time

For *PLOC* and *PLOC++*, a search radius of 16 was set as default, as it provides a good trade-off in terms of BVH quality and construction speed. The chunk size for the merged nearest neighbor search and BVH construction kernel is set to $1024 - 4 \times 16 = 960$ (1024 being the largest possible work group size). The $4 \times 16 = 64$ elements are due to 2×16 additional elements at the left and right borders of the chunk, which are needed to allow independent chunk processing (see Section 4.1). The switching threshold for the single work group construction kernel (see Section 4.4) was set to 8192. This keeps the number of required iterations most of the time below 50, compared to $70 - 180$ without the switch. For the *PLOC++ Two-Level* approach, the Morton code-based range extraction was set to extract a number of ranges which is two times 64, 64 being the minimal number of work groups required to fully utilize the GPU. In general, the output of the BVH builder is an uncompressed binary BVH which is, in a final step, converted into a specific layout consumed by hardware ray tracing units. This conversion step is independent from the underlying BVH build algorithm and therefore omitted from the performance measurements.

All performance measurements include all pre-processing steps such as scene bounding box computation, 64-bit Morton code generation, sorting of Morton codes, cluster initialization, etc. With respect to host vs. device timings, Meister et al. [2018a] report an average host-device overhead of 1.5× for *PLOC*. The overhead is caused by the per-kernel launch overhead and the number of kernel launches required. We measured the average host vs. device overhead on the Intel GPU to be on average 1.6×. The small difference is likely due to differences in the underlying GPU architecture, software layer and different kernel run-time costs. Figure 3 shows relative kernel run-times using device timings. *PLOC++*'s fusing and simplification of phases shows a significant costs reduction compared to *PLOC*, leading to an almost balanced cost between iterative and pre-processing kernels.

In the following, we will use host timings for all comparisons. Because of requiring just two kernel launches per iteration and using the single work group construction kernel for building the top of the BVH, *PLOC++* reduces the kernel launch overhead to an average of 1.4× (vs 1.6×), and even less than 1.1× for *PLOC++ Two-Level*. The reduced host vs. device overhead in combination with the faster chunk-based nearest neighbor search result in a total BVH construction speedup of $1.9 - 2.3\times$ compared to *PLOC* (see Figure 4), while *PLOC++ Two-Level* even reaches a speedup of $2 - 3\times$. *PLOC++ Two-Level* provides a speedup of $10 - 30\%$ over *PLOC++*.

In comparison to an optimized 4D LBVH builder [Vinkler et al. 2017] also based on 64-bit Morton codes (see Figure 4), one of the fastest BVH builders available, *PLOC* achieves a relative BVH build performance of $0.24 - 0.46\times$, while *PLOC++* reaches $0.56 - 0.92\times$ and *PLOC++ Two-Level* even $0.71 - 1.03\times$. An interesting outlier is the *Rungholt* scene which requires the lowest number of iterations of all scenes and therefore has the fastest *PLOC++* BVH build performance. For this scene,

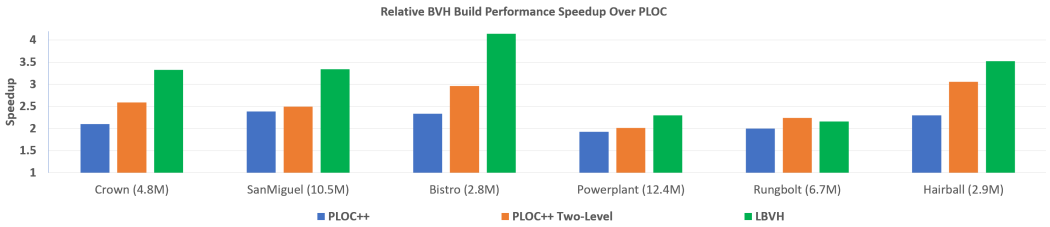

Fig. 4. The faster nearest neighbor search, the reduced number of kernel launches per iteration, and the single work group construction kernel allow *PLOC++* to outperform *PLOC* by 1.9 – 2.3×. The *PLOC++-Two-Level* extension, which relies on extracting an implicit two-level hierarchy from the Morton code, further reduces the kernel launch overhead, increasing the speedup factor of *PLOC++* over *PLOC* to 2 – 3×. In comparison to a *LBVH* builder, *PLOC* achieves just 0.23 – 0.4× of the *LBVH* build performance, while *PLOC++* reaches 0.51 – 0.86× and *PLOC++-Two-Level* even 0.66 – 0.97×.

PLOC++ Two-Level is slightly faster than the 4D LBVH builder, while for *Bistro* the number of required iterations is the highest, leading to the lowest BVH build performance. This makes *PLOC++* and in particular *PLOC++ Two-Level* a potential LBVH builder alternative as they provide on average a 5 – 18% better ray tracing performance (see Section 5.2). In case BVH build performance is more important than BVH quality, a smaller nearest neighbor search radius could be used (see Section 5.3) which is likely to affect quality but increase build performance of all *PLOC*-based approaches.

5.2 BVH Quality and Ray Tracing Performance

For comparing BVH quality and ray tracing performance during rendering, a simplified two bounce diffuse path tracer was used. At each intersection point, 64 random rays are cast, ensuring a high incoherence in the ray distribution. Our comparison baseline is a high-quality CPU-based binned-SAH BVH builder (16 bins per dimension). Figure 5 shows that the ray tracing performance (ray traversal and triangle intersection are fully hardware accelerated on the Intel® Alchemist-G10 GPU) is within a 0.9 – 1.05× range for both *PLOC++* and *PLOC++ Two-Level* , while the SAH quality is within a 0.9 – 1.09× range. For our example scenes, BVH quality and actual ray tracing performance roughly correlate. Interestingly, the range restriction of the nearest neighbor search for *PLOC++ Two-Level* and its impact on BVH structure does not affect the ray tracing performance much, as the difference compared to *PLOC++* is just a few percent.

For improving the quality of *PLOC++ Two-Level*, we stop the iterative BVH build process per range if the number of active cluster representatives is equal or less than the search radius (see Section 4.5) to allow for reducing the overlap in the top-level BVH construction phase. The following

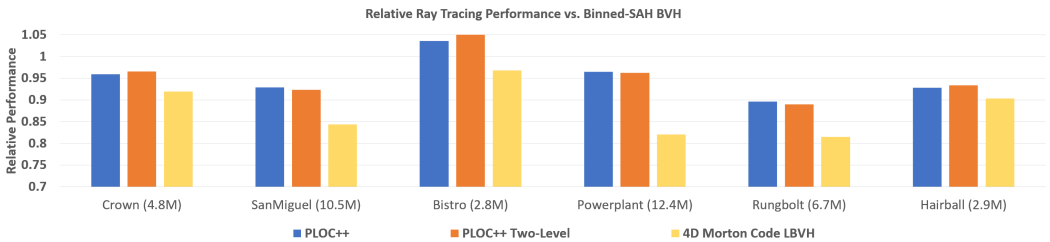

Fig. 5. Relative ray tracing performance for incoherent rays compared to a high-quality binned-SAH CPU BVH builder. *PLOC++* and *PLOC++-Two-Level* are within a 0.87 – 1.03× range, while *LBVH* is with 0.62 – 0.95×.

table shows the ray tracing performance benefit when using a search radius and stopping criteria of 16 compared to building the BVH per range until a single root node remains.

Scene	Crown	San Miguel	Bistro	Powerplant	Rungbolt	Hairball
Speedup	1%	2%	2%	9%	0%	7%

The biggest gains are visible for the *Powerplant* and *Hairball* scene, while the performance of the other scenes are largely unaffected. The likely reason for the gains in these two scenes is the increased overlap due to long and thin diagonal geometry.

5.3 Search Radius Impact

Increasing the search radius allows the nearest neighbor search to scan a larger window of potential candidates at the expense of making the search itself more expensive. Figure 6 shows relative BVH build and ray tracing performance with varying search radius. Increasing the search radius by 8 reduces the total build time by 5-10%. In terms of ray tracing performance, scenes with a large variation in triangle sizes and shapes, i.e. the *Powerplant* scene, benefit from a search radius of 16 or larger. At the same time a larger search radius than 16 does not necessarily mean better BVH quality and higher ray tracing performance, as for some scenes the performance slightly reduces in this case. Our chosen search radius of 16 seems to provide a sweet spot with respect to BVH build and ray tracing performance.

6 CONCLUSION AND FUTURE WORK

We proposed *PLOC++*, a novel version of the *PLOC* BVH construction algorithm, which is more efficient in terms of computation and memory requirements. The faster chunk-based nearest neighbor search, the fusion of kernels, and the single work group construction kernel allow *PLOC++* to outperform the original implementation by up to 2.3×, while *PLOC++ Two-Level* even increases the speedup up to 3×. We believe that our algorithm should directly be applicable to other GPUs, as most major GPU architectures both support large workgroups of 1024 work items and at least 32K of shared local memory per work group. Additionally, the implementation does not rely on any Intel specific hardware extension, hence a port to other programming environments should be straight forward. In the future we would like to explore further optimizations, e.g., replacing the caching of cluster representatives into shared local memory by loading the data into registers once and relying on register swizzles to exchange data across work items. Additionally, memory bandwidth could be saved by quantifying the clusters' bounding boxes and, in case of Morton code-extracted range partitions, the BVH quality could be improved by applying *re-braiding* based techniques [Benthin et al. 2017] to reduce spatial overlap. Another interesting approach would be

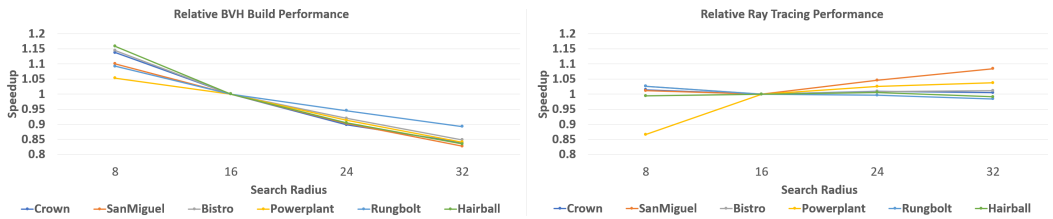

Fig. 6. Relative build time (left) and ray tracing performance (right) with varying search radius (normalized to performance using a search radius of 16). Increasing the search radius by 8 reduces build performance by 5-10%. Search radius of 8 reduces the ray tracing performance by 14% for the *Powerplant* scene, while a larger radius slightly increases performance for some scenes while reducing it for others.

to use *PLOC++* to improve quality of an already existing binary tree (e.g. Morton code-based) by rebuilding it in a single bottom-up pass. This would restrict the nearest neighbor search to the elements in a given sub-tree which will likely affect BVH quality. In terms of hardware acceleration, the implementation could directly benefit from offloading the merged nearest neighbor search and hierarchy construction phase to a dedicated hardware unit, similar to [Viitanen et al. 2018]. However, as sorting the Morton codes is starting to have similar run-time cost than the merged phase, a dedicated hardware unit for sorting key-value integer pairs would be beneficial as well.

ACKNOWLEDGMENTS

Model courtesy: Crown (Martin Lubich), SanMiguel (Guillermo M. Leal Llaguno), Bistro (Amazon Lumberyard), Powerplant (University of North Carolina), Rungholt (kescha), Hairball (NVIDIA Research).

REFERENCES

Timo Aila, Tero Karras, and Samuli Laine. 2013. On Quality Metrics of Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*. 101–108.

Carsten Benthin, Sven Woop, Ingo Wald, and Attila Áfra. 2017. Improved Two-Level BVHs using Partial Re-Braiding. In *Proceedings of High-Performance Graphics*.

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

Per Ganestam and Michael Doggett. 2016. SAH Guided Spatial Split Partitioning for Fast BVH Construction. *Computer Graphics Forum* (2016).

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.

Jakub Hendrich, Daniel Meister, and Jiří Bittner. 2017. Parallel BVH Construction Using Progressive Hierarchical Refinement. *Computer Graphics Forum (Proceedings of Eurographics)* 36, 2 (2017), 487–494.

Intel Corporation. 2021. oneAPI Programming Model. https://www.oneapi.com/

Intel Corporation. 2022. Intel Arc A-Series Graphics. https://www.intel.com/content/www/us/en/products/details/discrete-gpus/arc/arc-a-series.html

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.

Morgan McGuire. 2017. Computer Graphics Archive. https://casual-effects.com/data

Daniel Meister. 2018. PLOC Baseline implementation. https://github.com/meistdan/ploc/blob/master/README

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, 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).

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*.

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.

A APPENDIX

Absolute BVH build timings on a pre-production Intel Alchemist-G10 GPU with 32 Xe cores. [2]

Scene	Crown	San Miguel	Bistro	Powerplant	Rungbolt	Hairball
Triangles (M)	4.8	10.5	2.8	12.4	6.7	2.9
	PLOC					
Host Build Perf (Mprims/s)	100.8	104.6	71.2	159.9	156.3	84.3
	PLOC++					
Host (ms)	23.0	42.1	17.0	40.2	21.4	14.9
Device (ms)	16.1	33.8	11.2	33.3	16.6	9.9
Host Build Perf (Mprims/s)	211.8	249.7	166.1	308.3	312.6	193.8
	PLOC++ Two-Level					
Host (ms)	18.6	40.2	13.4	38.5	19.1	11.1
Device (ms)	16.8	38.0	12.0	36.2	17.3	9.6
Host Build Perf (Mprims/s)	261.4	261.1	210.9	321.6	349.6	260.8

[2]Performance varies by use, configuration and other factors. Learn more on the https://edc.intel.com/content/www/us/en/products/performance/benchmarks/overview/ Performance Index site. Results may vary. No product or component can be absolutely secure. Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy. Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade. Intel technologies may require enabled hardware, software or service activation. © Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.