Eurographics Symposium on Rendering - Experimental Ideas & Implementations (2015)
J. Lehtinen and D. Nowrouzezahrai (Editors)

GPU-based Out-of-Core HLBVH Construction

Mahmoud Zeidan, Taymoor Nazmy, and Mostafa Aref

Faculty of Computer and Information Sciences
Ain Shams University

Abstract

Recently the GPU has been used extensively in building indexing structures for moderately complex scenes that fit inside the GPU core. However, only few methods have been developed for constructing indexing structures for massive models larger than GPU memory. In this paper, we present an out-of-core HLBVH algorithm, a new method for constructing spatial hierarchies suitable for massive models that cannot fit into GPU device memory. A key insight of our method is how to bring and process out-of-core data blocks that do not fit into available device memory. Results show that our approach can compete with HLBVH hierarchy builder for large models on CPU. We also demonstrate the value of our algorithms in a GPU-based out-of-core path tracer that brings tree nodes and geometry into GPU core as needed, and efficiently achieve complex global illumination effects for models up to hundred million triangles.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Global illumination algorithms such as ray tracing, path tracing, and photon mapping [PH10] make an extensive use of bounding volume hierarchies (BVH) [Wal04, PH10] as spatial index structures for fast ray-primitive intersections. In the past few years, efficient methods for BVH construction have been proposed on both multicore CPU [Wal04, Wal07, PGDS09, SFD09] and manycore GPU [GPBG11, GPM11, PL10, LGS*09] for small and medium sized scenes [HSZ*11]. However, the request for more realism in visualization applications and movie production necessitate an efficient way to build indexing structures for large and massively large scenes [IBH11, PFHA10, KTO11, SBB*06].

In this paper, we focus on extending the recently developed HLBVH construction algorithm [Ape14, Kar12, GPM11, PL10, LGS*09] for massive models which are greatly larger than the available device memory. We present four main contributions. First, we introduce an out-of-GPU-core sorting procedure that can sort hundreds of millions of keys on GPU as long as the keys reside in main memory. Second, we present an out-of-GPU-core memory management strategy that iteratively loads and swaps data upon request between the host and device memory. Third, we develop a novel hierarchy emission strategy suitable for massively large scenes composed of several million triangles. Finally, we demonstrate the value of our hierarchy builder in an GPU-based out-of-core path tracer that produces complex global illumination effects for massively large models.

We follow partial breadth-first search (PBFS) order [HSZ*11] to construct tree hierarchy in three main stages; firstly, we iteratively load the sorted Morton codes into device memory and emit the first n levels using a large block descriptor for the root node [PL10], secondly, we use primitives sampling for out-of-GPU-core Morton codes and binary search [GPM11] to emit the following tree levels until we get a relatively large leaves with primitives less than a certain threshold, and finally, we iteratively load large leaves and their primitives and emit their corresponding treelets entirely inside the GPU.

In all subsequent sections, and unless mentioned otherwise, the term out-of-core is used to refer to out of GPU device memory.

DOI: 10.2312/sre.20151165

2. Related Work

2.1. Spatial Hierarchy Construction on GPU

Several methods have been proposed for constructing spatial hierarchy on GPU for grids [KS09], KD-trees [ZHWG08], and BVHs [Kar12,GPBG11,GPM11,PL10,LGS*09]. However, we concentrate on relevant work for BVH construction on GPU.

Lauterbach et al. [LGS*09] presented a fast BVH construction algorithm called Linear Bounding Volume Hierarchy (LBVH), which folds the construction process into two sorting rounds; the first over Morton codes generated by sampling primitives' centroids inside scene bounds, and the second over bit-difference indices (i.e. block-splits) between neighboring Morton codes. Later, Pantaleoni and Luebke [PL10] introduced the Hierarchical Linear Bounding Volume Hierarchy (HLBVH) algorithm, that produces the same hierarchy as of LBVH in less time. Pantaleoni and Luebke [PL10] replaced the first soring round of LBVH [LGS*09] by the relatively fast compress-sort-decompress (CSD) scheme [GL10]. In order to avoid the other sorting round, Pantaleoni and Luebke [PL10] emitted the tree hierarchy in several passes using block descriptors for block-splits, and prefix scans [HSO07, SHZO07] for nodes compaction.

While LBVH and HLBVH algorithms focused on Morton codes to find block-splits, and then mapping such block-splits into nodes' splits in the tree hierarchy. Garanzha et al. [GPM11] used binary search to find the block-split in each node, which directly maps to a node split. Garanzha et al. [GPM11] avoided the relatively slow several kernel launches in LBVH and HLBVH by emitting the entire tree hierarchy in a single kernel launch using running queues for input and output nodes on GPU.

Karras [Kar12] followed another direction to enhance the construction time of (H)LBVH by mapping Morton codes into binary radix tree (BRT). It was noted that hierarchy produced by (H)LBVH is identical to what produced by BRT if Morton codes are distinct, otherwise the hierarchy produced by BRT continue splitting leaves having more than one primitive in (H)LBVH until all leaves have exactly one primitive.

Most of the previous methods for BVH construction on GPU were limited by the available device memory, and required the whole scene, or the whole primitives to be loaded in advance inside the GPU [HSZ*11]. In this paper, we are interested in HLBVH construction for massive models which are larger than the available device memory. We begin by emitting the first n tree levels using a large block descriptor for the root node [PL10] to get a coarse tree hierarchy suitable for parallel processing. Then, we emit the following tree levels using binary search over a small set of out-of-core Morton codes until we reach a relatively large leaves having primitives less than a predefined threshold. Finally, we iteratively emit the remaining large leaves' treelets entirely in GPU using running queues and binary search for Morton codes splits [GPM11].

2.2. GPU based Global Illumination

In the past few years, the GPU has been used extensively to simulate global illumination effects using real-time ray tracing [ZHWG08], interactive photon mapping [WWZ*09], and path tracing [PBPP11]. Garanzha et al. [GBPG11] introduced a general rendering pipeline for GPU out-of-core data loading. Wang et al. [WHY*13] presented a GPU out-of-core method for many-lights rendering framework that renders scenes up to many million triangles using a large number of virtual point lights. We have a similar spirit to what presented by Garanzha et al. [GBPG11], and Wang et al. [WHY*13], and introduce a general memory paging strategy that iteratively loads the data needed upon request for hierarchy construction and various rendering tasks on GPU.

3. Our Approach

At the core of our algorithm is the use of a simple paging technique for data swapping between host and device memory. Our paging technique allows both data reading and writing for out-of-core data blocks that do not fit into GPU device memory. Generally, we follow the parallel rendering pipeline of *requesting, processing, and swapping* out-of-core data blocks on GPU [GBPG11].

Our algorithm begins by sampling primitives' centroids inside the scene bounding box to extract a 60-bit Morton code for each primitive. However, since we can not upload the whole primitives into GPU, we partition the primitives into buckets that can fit into available device memory, and extract the Morton codes of the primitives of each bucket separately. Next, we sort the Morton codes using an out-of-core key-value sorting algorithm. In order to sort a large array of keys that does not fit inside in-core memory, we split the keys into fixed-sized buckets so that we can sort each bucket in-core using a key-value sorting algorithm [HOS*07]. Then, we iteratively merge each pair of sorted buckets of size S_b to get a larger sorted bucket of size $2S_b$ [HR89], and recursively repeat the merge step until we get the whole array of keys sorted.

After sorting the Morton codes, we emit the entire tree hierarchy in three main stages: In the first stage, we build a coarse tree hierarchy for the upper levels by filling a large block descriptor [PL10] for the root node using the first n bits of Morton codes, and emit the corresponding treelet inside the GPU core. In the next stage, we build the hierarchy for the following tree levels in breadth-first search (BFS) order. We begin by partitioning the Morton codes into equally-sized pages and sample Morton codes at each page bounds. We use the sampled codes to approximate the search range for each node split by finding the the page containing node

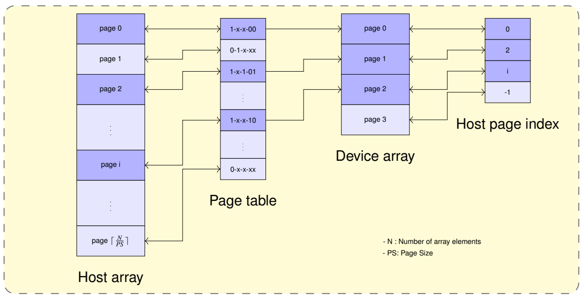

Figure 1: Out-of-core paging. In-core pages are highlighted in dark blue. The page table is used to link CPU pages to corresponding GPU pages. In each page table entry, the three most significant bits from left to right are: (1) the in-core flag, (2) the page request flag, and (3) the page modified flag, and the remaining bits are used to store page start address relative to GPU array start address. The host page index array is used to link GPU pages to the corresponding CPU pages.

split index using binary search over sampled codes. Then, we use our paging strategy to load the subset of pages containing the current nodes' splits, and use binary search inside the corresponding pages to get the exact nodes' splits. We continue BFS nodes splitting until each node has a relatively large number of primitives under a certain threshold. In our experiments, we set this threshold equal to the page size of Morton codes. In the final stage, we begin with the resulting large leaves from stage two, and iteratively upload subsets of these leaves and their primitives into GPU to emit their corresponding treelets entirely inside the GPU [GPM11].

4. Out-of-Core Paging

Page Table Structure In order to upload a large array that does not fit into available device memory, we use a simple paging technique. We partition the host array into fixed-sized pages, and partition the available device memory into pages of the same size. We use a page table to record CPU to GPU page referencing. For each page on CPU, we use a 32-bit page table entry, where bit 31 indicates whether the page is in-core or out-of-core, bit 30 is a page request flag, bit 29 is a page modified flag, and the remaining 29 bits are used to indicate the page offset relative to the device array start address (see Figure 1).

Out-of-Core Process, Request, and Swap Initially, we clear the page table by clearing the in-core flag of the entire page table. Then, we iteratively call the processing kernel several times until all page requests have been supplied and processing finishs. In the processing kernel, we perform CPU to GPU address translation of array elements. The address translation is very trivial; we extract the page index by dividing array index by page size, and get local offset inside the corresponding page using the remainder of this division. Given the page index, we examine the in-core flag of the corresponding page table entry, if the page exists inside the GPU, the kernel processes the array element, and arbitrarily set the modified bit flag if the array element has been modified, and needs to be written back to CPU. If the corresponding page is out of in-core memory, we invoke a page request using the page request flag.

After each kernel launch, we iterate over the page table and write back the modified pages and supply page requests that fit into available device memory. First, we check the modified page flag. If it has been set, we copy the corresponding page from the device to the host. Next, we check the page request flag. If it has been set, we copy the page from the host to the device and update the corresponding page table entry. We also keep an array that has a length equals to the number of pages on GPU. Inside this array we record CPU page index that resides in the corresponding GPU page or -1 if no page resides. During page uploading, we use this array to clear the in-core flag of the pages uploaded in the previous round, and update the entries of this array using the newly uploaded page-indices. We stop page uploading after we supply all pages requests, or reach the maximum number of page-uploads on GPU.

To reduce the data transfer overhead from CPU to GPU. We always keep two synchronized copies of the page table on the host and the device. After each kernel launch, we copy the entire page table from the device to the host to write back modified pages, and supply page requests at the host side. Before the next kernel launch, we copy back the page table from the host to the device to reflect the new memory mapping and use the device copy of the page table to raise new page requests, and new page modifications.

Mapping Multiple Arrays on GPU If we have more than one large array in a certain processing phase, we divide the available device memory among these arrays.

Iterative Kernels Launches Since each kernel may be launched several times during processing an out-of-core data, we assign a flag for each thread to mark thread processing status. At the beginning of the processing phase, this flag is set for all threads, and later when a thread receives its all data requests and performs data processing, it clears this flag to avoid processing in further kernel launches.

5. Out-of-Core HLBVH Construction

In this section, we explain in details the main processing stages for constructing an out-of-core HLBVH hierarchy. At first, we describe an out-of-core parallel sorting algorithm which is used to sort primitives' Morton codes. Then, using the sorted Morton codes, we present an out-of-core HLBVH hierarchy emission strategy suitable for massively large scenes.

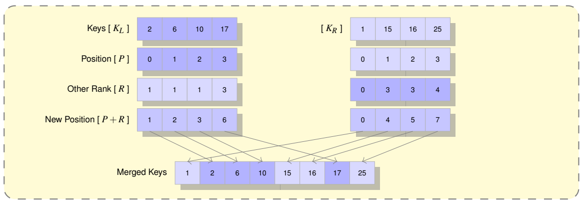

Figure 2: In-core merging using binary search.

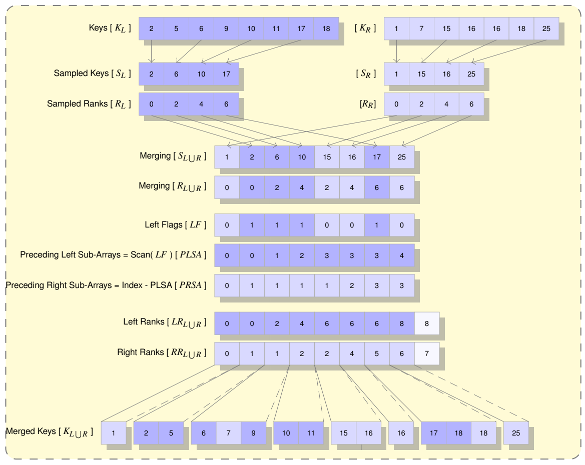
Keys [K_L] 2 5 6 9 10 11 17 18
[K_R] 1 7 15 16 16 18 25
Sampled Keys [S_L] 2 6 10 17
[S_R] 1 15 16 25
Sampled Ranks [R_L] 0 2 4 6
[R_R] 0 2 4 6
Merging [$S_{L\bigcup R}$] 1 2 6 10 15 16 17 25
Merging [$R_{L\bigcup R}$] 0 0 2 4 2 4 6 6
Left Flags [LF] 0 1 1 1 0 0 1 0
Preceding Left Sub-Arrays = Scan(LF) [$PLSA$] 0 0 1 2 3 3 3 4
Preceding Right Sub-Arrays = Index - PLSA [$PRSA$] 0 1 1 1 1 2 3 3
Left Ranks [$LR_{L\bigcup R}$] 0 0 2 4 6 6 6 8 8
Right Ranks [$RR_{L\bigcup R}$] 0 1 1 2 2 4 5 6 7
Merged Keys [$K_{L\bigcup R}$] 1 2 5 6 7 9 10 11 15 16 16 17 18 18 25

Figure 3: Out-of-core merging. The solid lines entering the array $K_{L\bigcup R}$ mark the inclusive start boundary, and the dashed lines mark the exclusive end boundary in the two ranks arrays ($LR_{L\bigcup R}$ and $RR_{L\bigcup R}$).

5.1. Out-of-Core Sorting

To sort Morton codes array that does not fit inside GPU memory, we begin by dividing the array into smaller N_b buckets, each of which has at most size S_b (in our experiments we set $S_b = 32$ M) so that each bucket is small enough to be sorted inside GPU using a fast parallel sorting algorithm [HOS*07]. Then, we perform $\lceil \log_2 N_b \rceil$ merge steps to sort the whole array. At the first merge step, we merge every two consecutive buckets of size S_b to get larger sorted buckets of size $2 \times S_b$. Next, we recursively repeat this merge step on the resulting buckets. After $\lceil \log_2 N_b \rceil$ merge steps we get whole array sorted.

Merging two sorted arrays which reside in in-core memory into a larger sorted array can be efficiently done in parallel. Each element at position P in an array finds its rank R in the other array using binary search. The key's new location in the output array will be at position $P+R$ (see Figure 2).

When the two arrays exceed the capacity of the device memory, we use Hagerup-Rüb algorithm [Sen10, HR89] for out-of-core merging. The out-of-core merge process is elaborated in Figure 3. We begin by sparsely sampling the b-th keys of both arrays into arrays S_L and S_R, and the associated ranks into arrays R_L and R_R. Then, we recursively merge the sampled keys and ranks using a key-value merge into arrays $S_{L\bigcup R}$ and $R_{L\bigcup R}$ respectively. For each key K_i in the array $S_{L\bigcup R}$, we extract two ranks; its ranks in left array of keys (LR_i), and its rank in the right array of keys (RR_i). One of these ranks is already known from the array of keys it was initially picked, and the other rank is extracted from the other array of keys using binary search over a small range of keys. We store the left and right ranks of each sampled key (i.e., LR_i and RR_i) into arrays $LR_{L\bigcup R}$ and $RR_{L\bigcup R}$ respectively.

However, finding keys' ranks in the other array of keys is not trivial, since each of the array of keys are stored out-of-core. We use an array of flags (i.e. *LF* array) for the sampled keys and its prefix scan to define a small search range for each sampled key in the array of keys it was not initially picked. Into the flags array we store 1 if the corresponding sampled key was picked from the left array, otherwise, we store 0. The scan of the flags array (i.e. *PLSA* array) records the number of sub-arrays preceding the current sampled key in the left array of keys, and by subtracting the scan of the flags array from the current sampled key index, we get an array (i.e. *PRSA* array) that records the number of sub-arrays preceding the current sampled key in the right array of keys. Given the number of sub-arrays preceding the current key, we restrict binary search for key's rank in the next keys' sub-array of the corresponding array of keys.

Corresponding ranges in the arrays $LR_{L\bigcup R}$ and $RR_{L\bigcup R}$ define contiguous and non-overlapped ranges in the left and the right arrays of keys respectively, which can be merged in-core together. Since we sample keys at every b-th position, no range will contain more that than b elements. In practice, we use several rounds to merge the keys according to the available device memory. At each round, we use the flags array and its scans (i.e. *LF*, *PLSA*, and *PRSA* arrays) to guide keys uploading into in-core memory. Once the keys uploaded, we fill the corresponding ranges in the ranks arrays (i.e., $LR_{L\bigcup R}$ and $RR_{L\bigcup R}$ arrays), and merge each pair of keys' ranges into an output array. We use values in the ranks arrays to define the start address, and the number of keys of each merge in the output array. As a final step, we copy the sorted output array to the host memory.

5.2. Out-of-core HLBVH Hierarchy Emission

As shown in Figure 4, we construct the entire tree hierarchy in three main stages:

- At the first stage, we emit the higher tree levels using a single block descriptor for root node [PL10].
- At the second stage, we begin with the leaves of root treelet, and emit the next levels using binary search over in-core sampled set of Morton codes, followed by binary search over out-of-core Morton codes employing our paging strategy to upload the required set of Morton codes into GPU. During this stage, we filter tree nodes having

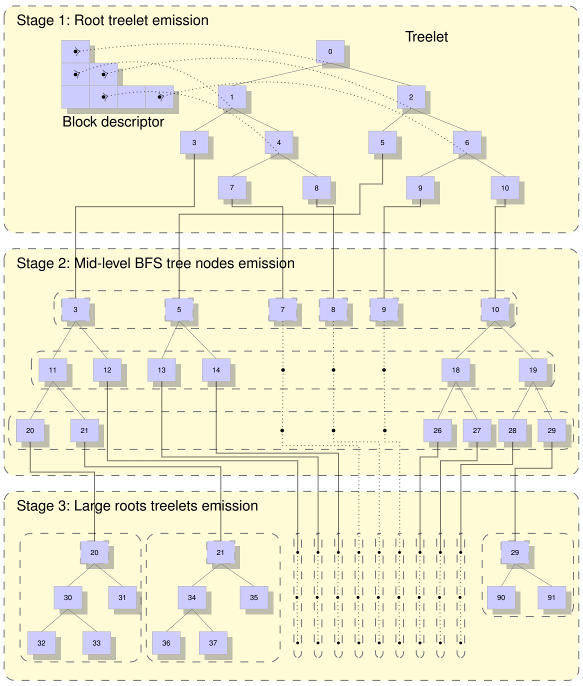
Stage 1: Root treelet emission
Treelet
Block descriptor
Stage 2: Mid-level BFS tree nodes emission
Stage 3: Large roots treelets emission

Figure 4: PBFS Out-of-core HLBVH hierarchy emission

triangles less than a predefined threshold to be split in in-core memory in the next stage.

- At the final stage, for all large leaves filtered in the previous stage, we iteratively load a subset of leaves and their data into in-core memory, and emit the corresponding treelets using binary search over Morton codes, and task queues for input and output nodes [GPM11].

5.2.1. Root Treelet Emission

We prepare a large block descriptor for the root node employing the firsts n bits of Morton codes to fill the split indices (in our experiments we set $n = 20$). As we are analyzing 20 bits, we need at most $2^{20}-1$ cells in the block descriptor. To fill the block descriptor, we partition the Morton codes into buckets that fit into available device memory, and iteratively upload the Morton codes buckets into GPU to fill split-indices between Morton codes, We modify the block descriptor a little bit, and store with each split index the two split Morton codes around it to be used later in the treelet emission (see Figure 5).

After filling the block descriptor we count the number of non-empty cells to prepare memory arena for the root treelet. Each non-empty cell will emit two child nodes. This makes the total number of nodes in the resulting treelet equals twice the number of non-empty cells in the block descriptor.

To complete the hierarchy, we need to determine the parent-child relationship of emitted nodes. We have noticed that the block descriptor stores the split indices sorted by split level. This allows us to build the parent-child relationship level by level using task-queues for input and output nodes at each level [GPM11]. For each input node, we find the split level l using the first bit plane by which the node's first, and last Morton codes differ. Then, we restrict the binary search in the block descriptor to level l, and find the split s having a primitive index that lies inside node's range, and link the input node to the two nodes emitted by split s, finally, we append the child nodes to the output queue for next level splitting. This process is highlighted with node 4 in Figure 5.

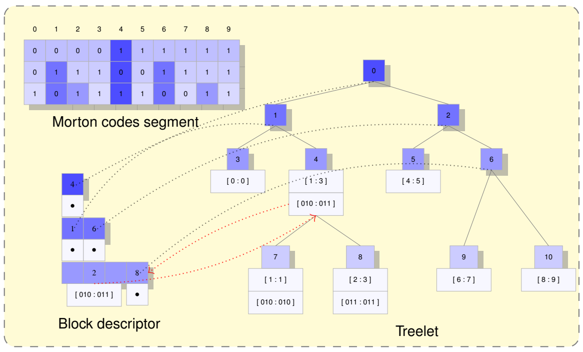

Figure 5: Root treelet emission. As an example node 4 which starts at primitive 1 and ends at primitive 3 uses the first and last Morton codes (i.e., [010 : 011]) to restrict the binary search at level 3 in the block descriptor for a primitive index that lies in its range (i.e., 2). Once the split found, node 4 is linked to child nodes corresponding to split 2 (i.e., node 7, and node 8). The ranges and Morton codes of child nodes are inherited from parent node, split index, and the two neighboring Morton codes around the split.

5.2.2. Mid-Level BFS Tree Nodes Emission

After emitting the first n tree levels in the previous stage, we begin with the leaves in root treelet, and build the several next tree levels in breadth-first search (BFS) order using an out-of-core binary search over Morton codes. First, we partition Morton codes into equally-sized pages, and sample the first and last code of each page. Following that, we split nodes level by level using task queues for input and output nodes [GPM11].

Each split round consist of two main steps (see Figure 6); in the first step, we launch a kernel that uses binary search over sampled Morton codes to find the corresponding Morton code page of node's split plane, and in the second step, we launch another kernel that uses binary search inside the corresponding pages of Morton codes to find the exact split index for each node. We use our paging strategy to iteratively upload the required pages for all input nodes into in-core memory. In this step, we keep track of the two Morton codes

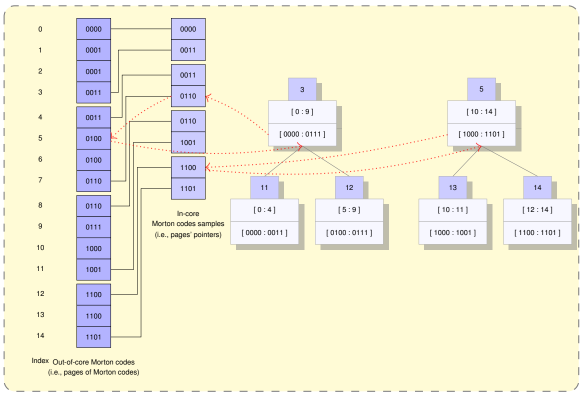

Figure 6: Mid-level BFS tree nodes emission using out-of-core binary search of Morton codes. The page size equals 4 for illustration purpose only. The red arrows illustrate the path from parent node, to sampled Morton codes, and optionally to Morton codes pages, and finally to child nodes.

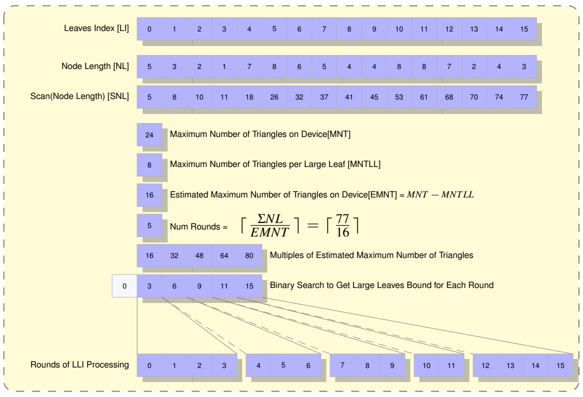
Leaves Index [LI] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Node Length [NL] 5 3 2 1 7 8 6 5 4 4 8 8 7 2 4 3
Scan(Node Length) [SNL] 5 8 10 11 18 26 32 37 41 45 53 61 68 70 74 77
24 Maximum Number of Triangles on Device[MNT]
8 Maximum Number of Triangles per Large Leaf [MNTLL]
16 Estimated Maximum Number of Triangles on Device[EMNT] = $MNT - MNTLL$
5 Num Rounds = $\lceil \frac{\Sigma NL}{EMNT} \rceil = \lceil \frac{77}{16} \rceil$
16 32 48 64 80 Multiples of Estimated Maximum Number of Triangles
0 3 6 9 11 15 Binary Search to Get Large Leaves Bound for Each Round
Rounds of LLI Processing 0 1 2 3 | 4 5 6 | 7 8 9 | 10 11 | 12 13 14 15

Figure 7: Estimating number of rounds for geometry splitting of large roots. The number of rounds depends on the number of triangles processed at each round according to the available device memory. We subtract the maximum triangles per node from the available triangles budget, since the previous round may not end at an integer multiple of available triangles budget.

around the split index to be used in the resulting child nodes. A special handling happens if node's split plane lies at the start of a certain page (e.g., node 5 in Figure 6). However, we have no need to upload any pages for this node, since our sampled Morton codes already record the Morton codes at the start and the end of each page. After finding nodes' splits, we split input nodes, and dump output nodes to CPU to be split in the next round. Exception happens with child nodes having primitives less than a predefined threshold (in our experiments we set this threshold to be equal the page size of Morton codes), which are filtered in a separate list to be split in in-core memory in the third stage.

5.2.3. Large Leaves Treelets Emission

At this stage, we have a list of relatively large-sized leaf nodes. For clarity, we refer to these nodes as large roots. Each of these large roots covers a relatively small spacial extent, and contains Morton codes that reside in at most two consecutive pages. Thus, for each large root, we emit its treelet entirely inside the GPU in a single kernel launch after uploading its corresponding Morton codes. Compared to global BFS emission in the previous stage, the advantages of this local BFS emission are two-fold:

- Less global memory traffic: by uploading at most two pages for each treelet, we avoid multiple page swapping occurring in global BFS node splitting.
- Ensure locality of reference during rendering: by emitting each treelet locally, and storing treelet nodes in nearby locations.

Splitting Nodes' Geometry In order to access the geometry of leaf nodes from the global scene data structures (i.e., sorted triangles' indices array, index buffer, and vertex buffer), we have to pass through several paging steps.

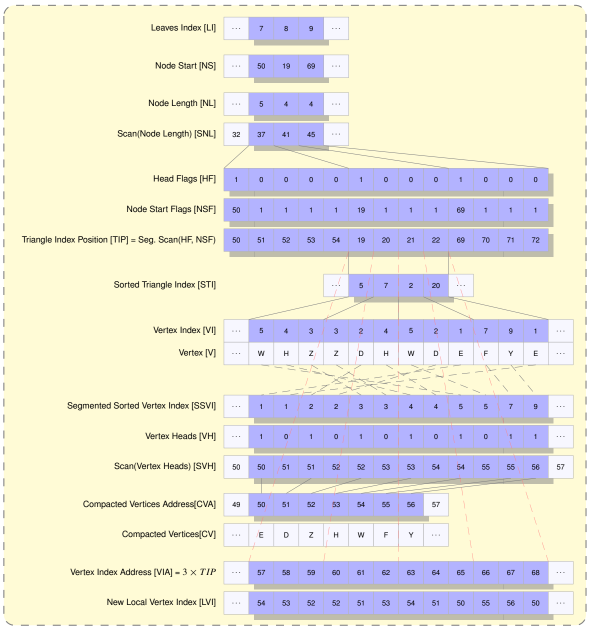
Leaves Index [LI] ··· 7 8 9 ···
Node Start [NS] ··· 50 19 69 ···
Node Length [NL] ··· 5 4 4 ···
Scan(Node Length) [SNL] 32 37 41 45 ···
Head Flags [HF] 1 0 0 0 0 1 0 0 0 1 0 0 0
Node Start Flags [NSF] 50 1 1 1 1 19 1 1 1 69 1 1 1
Triangle Index Position [TIP] = Seg. Scan(HF, NSF) 50 51 52 53 54 19 20 21 22 69 70 71 72
Sorted Triangle Index [STI] ··· 5 7 2 20 ···
Vertex Index [VI] ··· 5 4 3 3 2 4 5 2 1 7 9 1 ···
Vertex [V] ··· W H Z Z D H W D E F Y E ···
Segmented Sorted Vertex Index [SSVI] ··· 1 1 2 2 3 3 4 4 5 5 7 9 ···
Vertex Heads [VH] ··· 1 0 1 0 1 0 1 0 1 0 1 1 ···
Scan(Vertex Heads) [SVH] 50 50 51 51 52 52 53 53 54 54 55 55 56 57
Compacted Vertices Address[CVA] 49 50 51 52 53 54 55 56 57
Compacted Vertices[CV] ··· E D Z H W F Y ···
Vertex Index Address [VIA] = $3 \times TIP$ ··· 57 58 59 60 61 62 63 64 65 66 67 68 ···
New Local Vertex Index [LVI] ··· 54 53 52 52 51 53 54 51 50 55 56 50 ···

Figure 8: Geometry splitting of round 3 in Figure 7.

First, we have to load the sorted triangles' indices. Then, we load three vertex-indices of each triangle form the index buffer. And finally, we load the corresponding vertices of each vertex index from the vertex buffer. This may be a critical performance issue in ray tracing applications, since a certain node may refer to several distant blocks at the index buffer, and vertex buffer, and require several page loads even with small-sized nodes. Thus, for efficient ray tracing performance, we extract the geometry of current state roots from the global scene data structures, and relocate them compactly in a local index-vertex format.

To split the geometry of large roots, we estimate the number of triangles that can be processed in the available device memory, and accordingly calculate the number of nodes that can fit into GPU device (see Figure 7). To do so, we fill the length of each node into array *Node Length* and scan this array, and launch a parallel routine that performs a binary search in the scanned array for multiples of number of triangles that can be processed in the available device memory. Then, we initiate several rounds for geometry splitting for each set of roots bounded by indices returned from the binary search routine.

At each splitting round (see Figure 8), we calculate the number of triangles to be processed form the array *Node Length* and its scan. We prepare an array of head flags for triangles, and fill the head locations defined by the scan of *Node Length* array with $1s$ and fill other locations with $0s$. Then, we fill another array (*i.e., array TIP*) with triangles references to the sorted triangles indices of each node. To fill this array, we prepare another array (*i.e., array NSF*) which is initially filled with $1s$, and into corresponding locations of the scan of *Node Length* array, we store the corresponding node start index. Then, we invoke a segmented scan on the array *NSF* to get triangles references of each node (*i.e., array TIP*). Currently, we have a consecutive list of triangles references of each node referring to the sorted triangles indices array. Using our paging strategy, we load the triangles indices from the sorted array of triangles indices into array *STI*. Then, we use our paging strategy again to load the corresponding three vertex-indices of each triangle from the global index buffer into array *VI*.

At each node, some vertices may be referenced more than once, so we need to avoid duplicate loading of these vertices during rendering. In order to do so, we load the vertex-indices into 64 bits array, into this array we store the vertex index at the lowest 32 significant bits, and the store the node index at the highest 32 significant bits. Then, we sort this array using key-value sorting employing the 64 bits concatenated node index and vertex index pairs as keys. After sorting this array we have a segmented sorted array of vertex-indices, where the sorting process has been performed for each node separately. We invoke a parallel kernel that fills an array of flags (*i.e. array VH*). This kernel checks every key in the sorted array, and stores 1 into corresponding locations that have a key that differs from its predecessor, and stores 0 otherwise. Then, we perform a scan on this array into array *SVH*. The total length of these flags defines the exact number of compacted vertices of currently processed nodes, and using locations marked by node's first and last element index multiplied by three, we can calculate the start and the size of each node local vertex buffer.

Using the array *VH*, and its scan *SVH*, we load the compacted vertices once into local device memory using our paging strategy, and update the index buffer of each node to refer to the new location of vertices. As a final step, we copy back the compacted vertices to the host memory, and use our paging strategy to write back the sorted index buffer of currently processed nodes to host memory at locations derived from corresponding node's first and last triangles indices.

Hierarchy Emission At the beginning of this stage, we divide the available device memory into two partitions. In the first partition, we load large roots data such as nodes pointers, Morton codes, and other temporary data, and conservatively reserve a room for the the resulting treelets data (i.e., child nodes pointers, and AABBs). The second partition is used for paging out-of-core data such as triangles' data, and previously created nodes.

Since we can not process all the large roots inside the GPU in a single round, we have to divide these roots into smaller buckets suitable for the available device memory, and process each bucket separately. Each of the large roots requests a room in the device memory that is a function of the number of triangles inside the node (i.e., maximum treelet size is less than twice the number of triangles). Thus, we estimate the number of large roots processed in each round in way similar to Figure 7, excpet that we replace node's size with node's room size in the array *Node Length*.

At each processing round, we use our paging strategy to upload large roots' data, and corresponding primitives' data including Morton codes into GPU. Next, we let each thread block to process a single root, and emit its treelet in BFS order using shared memory and block synchronization in a manner similar to what explained by Garanzha et al. [GPM11]. After emitting the treelet, we preform a local AABB refitting on child nodes and the parent large root. As a final step after processing the current large roots, we compactly write back output treelets to CPU using a prefix scan of the exact sizes of all treelets, and update the current large roots data on CPU.

HLBVH Top-levels Refitting After processing all large roots, we perform AABB refitting on mid-level and high-level nodes employing our paging strategy to upload required child nodes into GPU upon request.

Scene	Num. of Triangles	Num. of Vertices	Num. of Nodes	Cum. Num. of Nodes in Stage 1 & 2
Buddha-Dragons	55 MT / 662 MB	29 MV / 344 MB	110 MN / 3.4 GB	21 KN / 22 KN
Armadillos	99 MT / 1188 MB	52 MV / 618 MB	151 MN / 4.7 GB	29 KN / 30 KN
Lucys	107 MT / 1284 MB	56 MV / 671 MB	214 MN / 6.7 GB	44 KN / 44 KN
MPI	70 MT / 844 MB	38 MV / 458 MB	128 MN / 4.0 GB	40 KN / 49 KN

Table 1: Tree statistics of our test scenes. First column lists the number of mega-triangles of each scene, and their storage size in Megabytes. Second column lists the number of mega-vertices after geometry splitting phase and their storage size in Megabytes. Third column lists the number of mega-nodes of the resulting HLBVH hierarchy and their storage size in Gigabytes. The forth column lists the cumulative number of nodes after the first and second stages respectively.

Scene	CPU Build Time	GPU Build Time	GPU Build Time without Stage 1
Buddha-Dragons	22 s (3507 ms)	11 s (1198 ms)	12 s
Armadillos	45 s (9187 ms)	20 s (2632 ms)	22 s
Lucys	48 s (6977 ms)	21 s (2857 ms)	26 s
MPI	27 s (4162 ms)	16 s (1929 ms)	15 s

Table 2: Build time statistics in seconds for our test scenes, the numbers inside brackets represent the sorting time of Morton codes in milliseconds.

Phase	Buddha-Dragons (11010 ms)	Armadillos (19787 ms)	Lucys (21444 ms)	MPI (16367 ms)
AABBs & Morton codes setup	1145 ms	2362 ms	2641 ms	1409 ms
Out-of-core sorting	1198 ms	2632 ms	2857 ms	1929 ms
Stage 1 (root treelet emission)	211 ms	362 ms	376 ms	298 ms
Stage 2 (mid-level nodes emission)	1 ms	1 ms	0.5 ms	81 ms
Stage 3-a (geometry splitting)	2738 ms	5186 ms	4689 ms	4146 ms
Stage 3-b (large roots treelets emission)	5717 ms	9244 ms	10880 ms	8504 ms

Table 3: Time breakdown for our test scenes, all numbers are in milliseconds.

6. Out-of-Core HLBVH Ray Traversal

We have noticed that HLBVH nodes created in the first and second stages are always small enough to be kept in device memory during the entire rendering process. For clarity, we refer to these nodes as top-nodes. Last column in Table 1 shows the number of top-nodes in our test scenes. The leaves of top-nodes are the large roots in the third stage, where each root references an out-of-core treelet. We employ a segment table for all out-of-core treelets, and perform out-of-core segmentation in a way similar to out-of-core paging as explained in Section 4.

In order to traverse a ray through HLBVH, we maintain two stacks for each ray, one for traversing the top-nodes, and the other is used when the ray hits an in-core treelet of a certain large root. Initially, and for all rays, we push the root node into the stack of top-nodes. Then, each ray performs an out-of-core ray tracing in several rounds.

During each round, the traversal kernel pops up the first node from the the stack of top-nodes, and visits top-nodes in the ordinary traversal way [AL09]. When the ray hits a large root, the kernel checks the corresponding treelet in-core flag. If the treelet resides in in-core memory, then the traversal kernel branches to another code segment that traverses the ray in the corresponding treelet using the other stack seeking for the nearest hit. If the corresponding treelet is out-of-core, then the ray stops traversal, pushes the current large root into the stack of top-nodes, and raises a treelet request. In future rounds, the thread of this ray continues checking the in-core flag of the requested treelet to resume traversal. When a ray finishes a large root treelet, it returns to the main code segment, and pops up the next node from the stack of top-nodes to resume traversal in top-nodes. The entire traversal ends when the stack of top-nodes is empty. At the host side after each traversal round, we supply treelets requests fitting in the available device memory, and update the segment table.

7. Results and Discussion

We implemented the above algorithms on an Intel Core i7-3770 3.40 GHz CPU with 32 GB of memory, and an NVIDIA GTX 670 graphics card having 2 GB of device memory under Microsoft Windows 8. All algorithms were implemented using NVIDIA CUDA. We used the CUDPP library [HOS*07] for data parallel primitives such as scan, segmented scan, and sorting.

We tested our algorithms with four test scenes (see Figure 9, and Table 1). The Buddha-Dragons scene contains one Buddha model (100 K triangles), and 8 Asian dragons (each 6.8 M triangles) inside a box. The Armadillos scene contains 300 armadillos (each 338 K triangles) on 30 shelves inside a box. The Lucys scene contains 4 Lucy models (each 26.8 M triangles) inside a box. And the MPI informatics building model (70 M triangles). In all scenes, we do not use any instancing for duplicate models, and all images were rendered at a resolution of 1024×1024.

For device memory allocation, similar to [HSZ*11], we allocate a small number of large buffers (i.e., two or three 500 MB buffers) on the device for the entire processing. The page size of all out-of-core data blocks was fixed at 16 K primitives. Each triangle is represented using three 32-bit integers for vertex-indices, where each vertex-index references three floats of vertex coordinates. Each tree node is stored in 32 bytes using structure of arrays (SoA) format.

We compare the construction time of our algorithm with a single-threaded CPU-based HLBVH construction algorithm that splits nodes using binary search to locate split planes as explained by Garanzha et al. [GPM11]. To sort Morton codes on CPU, we used the C++ Standard Template Library (STL) [SL95].

Table 2 lists the absolute build time of the HLBVH hierarchy for our test scenes. In all test scenes, our GPU hierarchy construction is faster than CPU-based hierarchy construction. It can be noted that Our HLBVH construction algorithm also scales well with the number primitives. We also report and compare our GPU-based out-of-core sorting against single-threaded STL sorting on CPU, and it can be shown that our sorting procedure is two to three times faster than CPU sorting for all test scenes. The last column in Table 2 lists the construction time of the HLBVH hierarchy

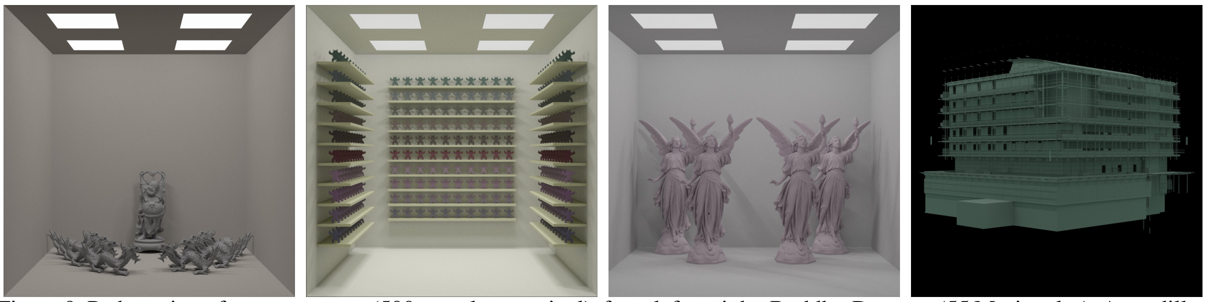
Figure 9: Path tracing of our test scenes (500 samples per pixel), from left to right: Buddha-Dragons (55 M triangles), Armadillos (99 M triangles), Lucys (107 M triangles), MPI building (70 M triangles).

Scene	Rendering Time	Average Memory Traffic / Frame	Time of Memory Traffic / Frame
Buddha-Dragons	31 s / 4 h 14 m	67 GB	28 s
Armadillos	23 s / 3 h 11 m	46 GB	19 s
Lucys	1 m 43 s / 14 h 21 m	228 GB	1 m 33 s
MPI	48 s / 6 h 47 m	105 GB	42 s

Table 4: Rendering time for our test scenes. In the first column, first numbers at each row represent the average render time of path-tracing one frame, and the second numbers at each row represent the total time for accumulating 500 frames of path-tracing test scenes as shown in Figure 9. Second column lists the average traffic of hierarchy and geometry uploading to GPU per frame. Last column lists the average time spent in uploading hierarchy and geometry data to GPU for one frame.

using only the second and third stages. It can be noted that the first stage saves a small amount of time for constructing the first tree levels in most scenes. This saving in time means that multiple uploading of Morton codes pages in the second stage consumes more traffic than uploading the whole Moron codes once to GPU.

We further analyzed the time spent in main processing phases in the hierarchy construction in Table 3. We found that most time was spent in geometry splitting and emitting large roots treelets in the third stage. This is due to the latency when copying data from CPU to GPU while paging triangles and nodes data, and the latency in copying back constructed treelets from GPU to CPU.

Path Tracing Performance We implemented a relatively simple and unoptimized path tracer with diffuse shading, where each ray undergoes at most 5 traversal bounces, and uses Russian roulette for early path termination. Figure 9 shows the path tracing results for our test scenes using 500 samples (i.e., paths) per pixel, and Table 4 shows the corresponding rendering time.

The results shown in Table 4 point out that most of the rendering time was consumed in page trffic between CPU and GPU. In our rendering pipline, each ray has an associated I/O cost related to the total size of all requested treelets. Currently, we do not use any mechanism for optimizing this I/O cost among rays.

The cost of uploading a treelet block into in-core memory is too much larger than uploading a ray. This may points to an interesting research problem that has been presented in another rendering framework [WHY*13]. In our rendering framework, we may either choose between loading another set of rays that hit a certain in-core treelet, or loading other requested treelets for currently in-core set of rays. Currently, we are working on finding the optimal data management strategy to minimize I/O cost that results from loading geometry and rays data blocks into GPU.

8. Conclusion and Future Work

We have presented a GPU-based out-of-core HLBVH construction algorithm suitable for massively large models. We also demonstrated the use of our HLBVH algorithm in high quality rendering of large scenes using out-of-core path tracing.

Currently, we are working on improving ray traversal, and path tracing performance. We believe that there is a large room for optimization. We also plan to investigate our rendering pipeline in other global illumination algorithms such as photon mapping [WWZ*09, ZHWG08], and progressive photon mapping [GG14, HJ10, HOJ08].

9. Acknowledgements

We would like to thank the anonymous reviewers for their thoughtful comments. We also want to thank Mohamed E. Fathy, and Karim Emara for their help in the proofreading. The Buddha, Asian Dragon, Armadillo, and Lucy models are courtesy of Stanford University. The MPI building is courtesy of Max-Planck-Institute for Informatics. Figure 5 was adapted from a similar one in [PL10].

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 145–149. 8

[Ape14] APETREI C.: Fast and simple agglomerative LBVH construction. In *Computer Graphics and Visual Computing (CGVC)* (2014), Borgo R., Tang W., (Eds.), The Eurographics Association. 1

[GBPG11] GARANZHA K., BELY A., PREMOZE S., GALAKTIONOV V.: Out-of-core GPU ray tracing of complex scenes. In *ACM SIGGRAPH 2011 Talks* (New York, NY, USA, 2011), SIGGRAPH '11, ACM, pp. 21:1–21:1. 2

[GG14] GÜNTHER T., GROSCH T.: Distributed out-of-core stochastic progressive photon mapping. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 154–166. 9

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. 2

[GPBG11] GARANZHA K., PREMOŽE S., BELY A., GALAKTIONOV V.: Grid-based SAH BVH construction on a GPU. *Vis. Comput. 27*, 6-8 (June 2011), 697–706. 1, 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 59–64. 1, 2, 3, 5, 7, 8

[HJ10] HACHISUKA T., JENSEN H. W.: Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches* (2010), ACM, p. 54. 9

[HOJ08] HACHISUKA T., OGAKI S., JENSEN H. W.: Progressive photon mapping. *ACM Transactions on Graphics (TOG) 27*, 5 (2008), 130. 9

[HOS*07] HARRIS M., OWENS J., SENGUPTA S., ZHANG Y., DAVIDSON A.: CUDPP: CUDA data parallel primitives library, 2007. 2, 4, 8

[HR89] HAGERUP T., RÜB C.: Optimal merging and sorting on the erew pram. *Inf. Process. Lett. 33*, 4 (Dec. 1989), 181–185. 2, 4

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, Aug. 2007. 2

[HSZ*11] HOU Q., SUN X., ZHOU K., LAUTERBACH C., MANOCHA D.: Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (2011), 466–474. 1, 2, 8

[IBH11] IZE T., BROWNLEE C., HANSEN C. D.: Real-time ray tracer for visualizing massive models on a cluster. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization* (2011), EG PGV'11, Eurographics Association, pp. 61–69. 1

[Kar12] KARRAS T.: 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* (2012), EGGH-HPG'12, Eurographics Association, pp. 33–37. 1, 2

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, pp. 23–28. 2

[KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. In *Proceedings of the Twenty-second Eurographics Conference on Rendering* (2011), EGSR'11, Eurographics Association, pp. 1353–1360. 1

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 2

[PBPP11] PAJOT A., BARTHE L., PAULIN M., POULIN P.: Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. *Computer Graphics Forum 30*, 2 (2011), 315–324. 2

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: Pantaray: Fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph. 29*, 4 (July 2010), 37:1–37:10. 1

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object partitioning considered harmful: space subdivision for BVHs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 15–22. 1

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2010. 1

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *In Proceedings of High Performance Graphics'10* (2010), pp. 87–95. 1, 2, 4, 9

[SBB*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S. G.: An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 19–26. 1

[Sen10] SENGUPTA S.: *Efficient Primitives and Algorithms for Many-core architectures*. PhD thesis, University of California Davis, 2010. 4

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 7–13. 1

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106. 2

[SL95] STEPANOV A., LEE M.: *The standard template library*, vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995. 8

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 1

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

[WHY*13] WANG R., HUO Y., YUAN Y., ZHOU K., HUA W., BAO H.: GPU-based out-of-core many-lights rendering. *ACM Trans. Graph. 32*, 6 (Nov. 2013), 210:1–210:10. 2, 9

[WWZ*09] WANG R., WANG R., ZHOU K., PAN M., BAO H.: An efficient GPU-based approach for interactive global illumination. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers* (New York, NY, USA, 2009), ACM, pp. 1–8. 2, 9

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (2008), 1–11. 2, 9