High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Maximizing Parallelism in the Construction of BVHs, Octrees, and *k*-d Trees

Tero Karras

NVIDIA Research

Abstract

A number of methods for constructing bounding volume hierarchies and point-based octrees on the GPU are based on the idea of ordering primitives along a space-filling curve. A major shortcoming with these methods is that they construct levels of the tree sequentially, which limits the amount of parallelism that they can achieve. We present a novel approach that improves scalability by constructing the entire tree in parallel. Our main contribution is an in-place algorithm for constructing binary radix trees, which we use as a building block for other types of trees.

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

1. Introduction

In the recent years, general-purpose GPU computing has given rise to a number of methods for constructing bounding volume hierarchies (BVHs), octrees, and *k*-d trees for millions of primitives in real-time. Some methods aim to maximize the quality of the resulting tree using the surface area heuristic [DPS10], while others choose to trade tree quality for increased construction speed [LGS*09, PL10, GPM11].

The right quality vs. speed tradeoff depends heavily on the application. Tree quality is usually preferable in ray tracing [AL09] where the same acceleration structure is often reused for millions of rays. Broad-phase collision detection [Eri04] and particle interaction [YB11] in real-time physics represent the other extreme, where construction speed is of primary importance—the acceleration structure has to be reconstructed on every time step, and the number of queries is usually fairly small. Furthermore, certain applications, such as voxel-based global illumination [CNS*11] and surface reconstruction [ZGHG11], specifically rely on regular octrees and *k*-d trees, where tree quality is fixed.

The main shortcoming with existing methods that aim to maximize construction speed [GPM11, ZGHG11] is that they generate the node hierarchy in a sequential fashion, usually one level at a time. This limits the amount of parallelism that they can achieve at the top levels of the tree, and can lead to serious underutilization of the parallel cores. The sequential processing is already a bottleneck with small workloads on current GPUs, which require tens of thousands of independent parallel threads to fully utilize their computing power. The problem can be expected to become even more significant in the future as the number of parallel cores keeps increasing. Another implication of sequential processing is that the existing methods output the hierarchy in a breadth-first order, even though a depth-first order would usually be preferable considering data locality and cache hit rates.

In this paper, we introduce a fast method for constructing BVHs, octrees, and *k*-d trees so that the overall performance scales linearly with the number of available cores (Figure 1) and the resulting data structure is always in a strict depth-first order. We start by presenting a novel in-place algorithm for constructing binary radix trees in a fully data-parallel fashion, and then show how the algorithm can be used as a building block for efficiently constructing other types of trees.

2. Background

Lauterbach et al. [LGS*09] were the first to present a parallel method for constructing so-called *linear BVHs* by ordering the input primitives along a space-filling curve. The idea is to assign a Morton code for each primitive, sort the Morton codes, and generate a node hierarchy where each subtree corresponds to a linear range of sorted primitives. The sorting effectively groups the primitives so that ones close to each other in 3D end up close to each other in the resulting tree.

The Morton code for a given point contained within the 3D unit cube is defined by the bit string $X_0Y_0Z_0X_1Y_1Z_1\ldots$

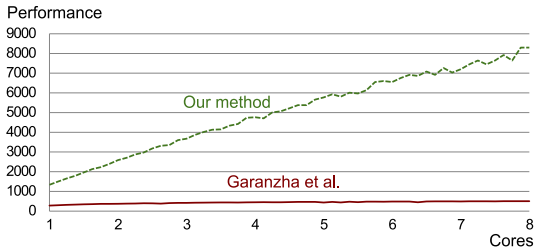

Figure 1: *Performance of BVH hierarchy generation for Stanford Dragon (871K triangles). The y-axis corresponds to millions of primitives per second and the x-axis to the number of parallel cores relative to GTX 480. The solid red line indicates the fastest existing method by Garanzha et al. [GPM11], which is dominated by the top levels of the tree where the amount of parallelism is very limited. Our method, indicated by the dotted green line, parallelizes over the entire tree and scales linearly with the number of cores.*

where the x coordinate of the point is represented as $0.X_0X_1X_2\ldots$, and similarly for y and z coordinates. The Morton code of an arbitrary 3D primitive can be defined in terms of the centroid of its axis-aligned bounding box (AABB). In practice, the Morton codes can be limited to 30 or 63 bits in order to store them as 32-bit or 64-bit integers, respectively.

The algorithm for generating BVH node hierarchy was subsequently improved by Pantaleoni and Luebke [PL10] and Garanzha et al. [GPM11]. Garanzha et al. generate one level of nodes at a time, starting from the root. They process the nodes on a given level in parallel, and use binary search to partition the primitives contained within each node. They then enumerate the resulting child nodes using an atomic counter, and subsequently process them on the next round.

Since these methods are targeted for real-time ray tracing, they are each accompanied with a high-quality construction algorithm to allow different quality vs. speed tradeoffs. The idea is to use the high-quality algorithm for a relatively small number of nodes near the root, and the fast algorithm for the rest of the tree. While we do not explicitly consider such hybrid methods in this paper, we believe that our approach is general enough to be combined with any appropriate high-quality algorithm in the same fashion.

Zhou et al. [ZGHG11] also applied the idea of using Morton codes to construct octrees in the context of surface reconstruction. Instead of generating the hierarchy in a top-down fashion, they start with the leaf nodes and perform a series of parallel compaction operations to determine their ancestors, one level at a time.

Binary radix trees. Given a set of n keys $k_0,\ldots,k_{n-1}$ represented as bit strings, a binary radix tree (also called a Patricia tree) is a hierarchical representation of their common prefixes. The keys are represented by the leaf nodes, and each internal node corresponds to the longest common prefix shared by the keys in its respective subtree (Figure 2).

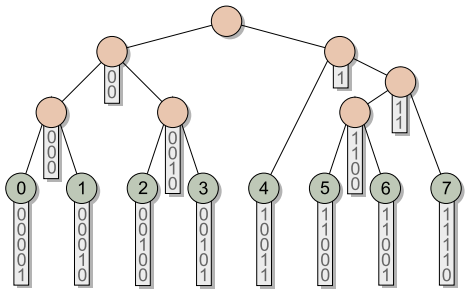

Figure 2: *Ordered binary radix tree. Leaf nodes, numbered 0–7, store a set of 5-bit keys in lexicographical order, and the internal nodes represent their common prefixes. Each internal node covers a linear range of keys, which it partitions into two subranges according to their first differing bit.*

In contrast to a prefix tree, which contains one internal node for every common prefix, a radix tree is compact in the sense that it omits nodes with only one child. Therefore, every binary radix tree with n leaf nodes contains exactly $n-1$ internal nodes. Duplicate keys require special attention—this is discussed in Section 4.

We will only consider ordered trees, where the children of each node—and consequently the leaf nodes—are in lexicographical order. This is equivalent to requiring that the sequence of keys is sorted, which enables representing the keys covered by each node as a linear range $[i,j]$. Using $\delta(i,j)$ to denote the length of the longest common prefix between keys k_i and k_j, the ordering implies that $\delta(i',j') \geq \delta(i,j)$ for any $i',j' \in [i,j]$. We can thus determine the prefix corresponding to a given node by comparing its first and last key—the other keys are guaranteed to share the same prefix.

In effect, each internal node partitions its keys according to their first differing bit, i.e. the one following $\delta(i,j)$. This bit will be zero for a certain number of keys starting from k_i, and one for the remaining ones until k_j. We call the index of the last key where the bit is zero a *split position*, denoted by $\gamma \in [i,j-1]$. Since the bit is zero for k_γ and one for $k_{\gamma+1}$, the split position must satisfy $\delta(\gamma,\gamma+1) = \delta(i,j)$. The resulting subranges are given by $[i,\gamma]$ and $[\gamma+1,j]$, and are further partitioned by the left and right child node, respectively.

In the figure, the root corresponds to the full range of keys, $[0,7]$. Since k_3 and k_4 differ at their first bit, the range is split at $\gamma = 3$, resulting in subranges $[0,3]$ and $[4,7]$. The left child further splits $[0,3]$ at $\gamma = 1$ based on the third bit, and the right child splits $[4,7]$ at $\gamma = 4$ based on the second bit.

3. Parallel Construction of Binary Radix Trees

A naïve algorithm for constructing a binary radix tree would start from the root, find the first differing bit, create the child nodes, and process each child recursively. This approach is inherently sequential—even though we know there are going to be $n-1$ internal nodes in the end, we have no knowl-

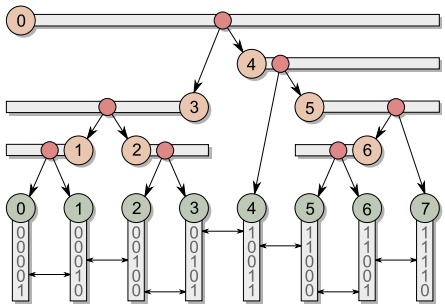

Figure 3: *Our node layout for the tree of Figure 2. Each internal node has been assigned an index between 0–6, and aligned horizontally with a leaf node of the same index. The range of keys covered by each node is indicated by a horizontal bar, and the split position, corresponding to the first bit that differs between the keys, is indicated by a red circle.*

edge of which keys they cover before having processed their ancestors. Our key insight in enabling parallel construction is to establish a connection between node indices and keys through a specific tree layout. The idea is to assign indices for the internal nodes in a way that enables finding their children without depending on earlier results.

Let us assume that the leaf nodes and internal nodes are stored in two separate arrays, L and I, respectively. We define our node layout so that the root is located at I_0, and the indices of its children—as well as the children of any internal node—are assigned according to its respective split position. The left child is located at I_γ if it covers more than one key, or at L_γ if it is a leaf. Similarly, the right child is located at $I_{\gamma+1}$ or $L_{\gamma+1}$. The layout is illustrated in Figure 3.

An important property of this particular layout is that the index of every internal node coincides with either its first or its last key. This follows by construction—the root is located at the beginning of its range $[0,n-1]$, the left child of any internal node is located at the end of its range $[i,\gamma]$, and the right child is located at the beginning of its range $[\gamma+1,j]$.

Algorithm. In order to construct a binary radix tree, we need to determine the range of keys covered by each internal node, as well as its children. The above property readily gives us one end of the range, and we will show how the other end can be found efficiently by looking at the nearby keys. The children can then be identified by finding the split position, by virtue of our node layout.

Pseudocode for the algorithm is given in Figure 4. We process each internal node I_i in parallel, and first determine the "direction" of its range by looking at the neighboring keys k_{i-1}, k_i, k_{i+1}. We denote the direction by d, so that $d=+1$ indicates a range beginning at i and $d=-1$ a range ending at i. Since every internal node covers at least two keys, we know that k_i and k_{i+d} must belong to I_i. We also know that k_{i-d} belongs to a sibling node I_{i-d}, since siblings are always located next to each other in our layout.

```
1:  for each internal node with index i ∈ [0, n−2] in parallel
2:      // Determine direction of the range (+1 or -1)
3:      d ← sign(δ(i, i+1) − δ(i, i−1))
4:      // Compute upper bound for the length of the range
5:      δ_min ← δ(i, i−d)
6:      l_max ← 2
7:      while δ(i, i+l_max·d) > δ_min do
8:          l_max ← l_max·2
9:      // Find the other end using binary search
10:     l ← 0
11:     for t ← {l_max/2, l_max/4, ..., 1} do
12:         if δ(i, i+(l+t)·d) > δ_min then
13:             l ← l+t
14:     j ← i+l·d
15:     // Find the split position using binary search
16:     δ_node ← δ(i, j)
17:     s ← 0
18:     for t ← {⌈l/2⌉, ⌈l/4⌉, ..., 1} do
19:         if δ(i, i+(s+t)·d) > δ_node then
20:             s ← s+t
21:     γ ← i+s·d+min(d, 0)
22:     // Output child pointers
23:     if min(i, j) = γ then left ← L_γ else left ← I_γ
24:     if max(i, j) = γ+1 then right ← L_{γ+1} else right ← I_{γ+1}
25:     I_i ← (left, right)
26: end for
```

Figure 4: *Pseudocode for constructing a binary radix tree. For simplicity, we define that* $\delta(i,j)=-1$ *when* $j\notin[0,n-1]$.

Now, the keys belonging to I_i share a common prefix that must be different from the one in the sibling by definition. This implies that a lower bound for the length of the prefix is given by $\delta_{\min}=\delta(i,i-d)$, so that $\delta(i,j)>\delta_{\min}$ for any k_j belonging to I_i. We can satisfy this condition by comparing $\delta(i,i-1)$ with $\delta(i,i+1)$, and choosing d so that $\delta(i,i+d)$ corresponds to the larger one (line 3).

We use the same reasoning to find the other end of the range by searching for the largest l that satisfies $\delta(i,i+ld)>\delta_{\min}$. We first determine a power-of-two upper bound $l_{\max}>l$ by starting from 2 and increasing the value exponentially until it no longer satisfies the inequality (lines 6–8). Once we have the upper bound, we find l using binary search in the range $[0,l_{\max}-1]$. The idea is to consider each bit of l in turn, starting from the highest one, and set it to one unless the new value would fail to satisfy the inequality (lines 10–13). The other end of the range is then given by $j=i+ld$.

$\delta(i,j)$ tells us the length of the prefix corresponding to I_i, which we shall denote by δ_{node}. We can, in turn, use this to find the split position γ by performing a similar binary search for largest $s\in[0,l-1]$ satisfying $\delta(i,i+sd)>\delta_{\text{node}}$ (lines 17–20). If $d=+1$, γ is then given by $i+sd$, as this is the highest index belonging to the left child. If $d=-1$, we have to decrement the value to account for the inverted indexing.

Given i, j, and γ, the children of I_i cover the ranges $[\min(i,j),\gamma]$ and $[\gamma+1,\max(i,j)]$. For each child, we compare the beginning and end of its range to see whether it is a

leaf, and then reference the corresponding node at index γ or $\gamma+1$ in accordance with our layout (lines 23–24).

GPU implementation. The algorithm can be implemented on a GPU as a single kernel launch, where each thread is responsible for one internal node. Assuming that the length of the keys is fixed, $\delta(i,j)$ can be evaluated efficiently by computing logical XOR between the two keys and counting the leading zero bits in the resulting integer. This can be done either by using the `__clz()` compiler intrinsic in CUDA, or by converting the integer to floating point and then calculating $31 - \lfloor \log_2 x \rfloor$. When searching for $l_{\max}$ on lines 5–8, we have found that it is beneficial to start from a larger number, e.g. 128, and multiply the value by 4 instead of 2 after each iteration to reduce the total amount of work.

Time complexity. For a node covering l keys, each of the three loops executes at most $\lceil \log_2 l \rceil$ iterations. Since $l \leq n$ for all $n-1$ internal nodes, this leads to a worst-case time complexity of $\mathcal{O}(n \log n)$ for the entire tree. The worst case is only realized when the height of the tree grows proportional to n. However, in practice the height is limited by the length of the keys, and is often proportional to $\log n$. For height h, we can obtain a tighter bound of $\mathcal{O}(n \log h)$ because most of the subtrees are guaranteed to be small if the tree is shallow.

4. BVHs, Octrees, and k-d Trees

BVHs. Following in the footsteps of Lauterbach et al. [LGS*09], we construct a BVH for a set of 3D primitives as follows: (1) assign a Morton code for each primitive according to its centroid, (2) sort the Morton codes, (3) construct a binary radix tree, and (4) assign a bounding box for each internal node. We contribute to steps 3–4.

If the Morton codes of all primitives are unique, it is easy to see that the binary radix tree is identical in structure to the corresponding linear BVH—identifying the common prefixes between the Morton codes is equivalent to bucketing the primitives recursively according to each bit. The case of duplicate Morton codes has to be handled explicitly, since our construction algorithm relies on the keys being unique. We accomplish this by augmenting each key with a bit representation of its index, i.e. $k'_i = k_i \oplus i$, where $\oplus$ indicates string concatenation. In practice, there is no need to actually store the augmented keys—it is enough to simply use i and j as a fallback if $k_i = k_j$ when evaluating $\delta(i,j)$.

Previous methods for linear BVHs calculate the bounding boxes sequentially in a bottom-up fashion, relying on the fact that the set of nodes located on each level is known a priori. We adopt a different approach where the paths from leaf nodes to the root are processed in parallel. Each thread starts from one leaf node and walks up the tree using parent pointers that we record during radix tree construction. We track how many threads have visited each internal node using atomic counters—the first thread terminates immediately while the second one gets to process the node. This way, each node is processed by exactly one thread, which leads to $\mathcal{O}(n)$ time complexity. The number of global atomics can be reduced by using faster shared memory atomics whenever we detect that all the leaves covered by a given internal node are being processed by the same thread block.

Octrees. To construct an octree for a set of points, we observe that each $3k$-bit prefix of a given Morton code maps directly to an octree node at level k. We can enumerate these prefixes by looking at the edges of a corresponding binary radix tree—an edge connecting a parent with a prefix of length δ_{parent} to a child with a prefix of length δ_{child} represents all subprefixes of length $\delta_{\text{parent}}+1, \ldots, \delta_{\text{child}}$. Out of these, $\lfloor \delta_{\text{child}}/3 \rfloor - \lfloor \delta_{\text{parent}}/3 \rfloor$ are divisible by 3. We evaluate these counts during radix tree construction, and then perform a parallel prefix sum to allocate the octree nodes. The parents of the octree nodes can then be found by looking at the immediate ancestors of each radix tree node.

The processing thus consists of seven steps: (1) calculate Morton codes for the points, (2) sort the Morton codes, (3) identify duplicates, i.e. points falling within the same leaf node, by comparing each pair of adjacent Morton codes, (4) remove the duplicates using parallel compaction, (5) construct a binary radix tree, (6) perform parallel prefix sum to allocate the octree nodes, and (7) find the parent of each node. We contribute to steps 5–7.

k-d trees. The radix tree produced by step 5 above can be interpreted directly as a k-d tree over the points. Every internal node partitions the points according to the next bit in the Morton codes after their common prefix, which is equivalent to classifying them on either side of an axis-aligned plane in 3D. A prefix of length δ corresponds to a plane perpendicular to the dth main axis, where $d = \delta \bmod 3$. The position of the plane is given by $0.B_d B_{d+3} \ldots B_{\delta-3} 1$, where B_i represents the ith bit of the prefix.

5. Results

We implemented our algorithms using CUDA 4.0 on GeForce GTX 480, installed in a PC with 2.80 GHz Intel Core i7 CPU running Windows 7. We used the efficient parallel primitives by Merrill and Grimshaw [MG10] for sorting and compaction. For comparison, we also implemented the methods by Garanzha et al. [GPM11], without top-level SAH splits, and Zhou et al. [ZGHG11]. We used 30-bit Morton codes in all benchmarks for maximum performance.

To gauge scalability, we simulated larger GPUs with N times as many cores and N times the memory bandwidth by breaking each kernel launch into N equal sized groups of thread blocks. Assuming that each group was executed concurrently in its own portion of the simulated GPU, the execution time would be dictated by the longest running group. We thus timed each group separately, and reported the largest execution time.

Table 1 shows a breakdown of BVH construction times

Scene	Cores	Common Eval	Common Sort	Build Our	Build Prev	AABB Our	AABB Prev
Fairy Forest (174K tris)	1×	0.05	0.56	**0.15**	**1.88**	0.23	0.29
	2×	0.03	0.33	**0.09**	**1.75**	0.13	0.22
	4×	0.02	0.30	**0.05**	**1.68**	0.08	0.19
Conference Room (283K tris)	1×	0.08	0.78	**0.24**	**1.93**	0.35	0.38
	2×	0.04	0.51	**0.13**	**1.72**	0.19	0.26
	4×	0.03	0.31	**0.08**	**1.58**	0.12	0.20
Stanford Dragon (871K tris)	1×	0.22	1.67	**0.65**	**3.14**	1.10	1.03
	2×	0.12	1.09	**0.34**	**2.38**	0.57	0.60
	4×	0.06	0.64	**0.18**	**1.97**	0.30	0.38
Turbine Blade (1.77M tris)	1×	0.45	2.73	**1.28**	**4.73**	2.10	1.77
	2×	0.23	1.63	**0.65**	**3.19**	1.07	0.96
	4×	0.12	1.08	**0.34**	**2.37**	0.56	0.55

Table 1: *BVH construction times for our method (Our) and Garanzha et al. (Prev) in milliseconds, with varying number of cores relative to GTX 480. The processing consists of Morton code evaluation (Eval), sorting (Sort), hierarchy generation (Build), and bounding box calculation (AABB).*

for a set of test scenes. With 1× cores, the performance of our radix tree construction is 4–13× compared to the hierarchy generation of Garanzha et al. The ratio is highest with small scenes, where the lack of parallelism in the comparison method is the most pronounced. The effect of increasing the number of cores is illustrated in Figure 5. The time required to evaluate and sort the Morton codes is the same for both methods, and decreases gradually with increasing parallelism. Hierarchy generation and AABB calculation consistently amount to 40–50% of the total time in our method, whereas the comparison method becomes severely bottlenecked by the hierarchy generation as the number of cores increases.

Table 2 shows a similar breakdown of octree construction times, using centroids of the triangles as the input points. Our method consistently performs and scales better than the comparison method, but the difference is not as pronounced as with BVHs. The main reason is that octrees have significantly fewer levels, which reduces the sequential processing overhead in the comparison method.

Discussion. Our parallel radix tree construction algorithm is a powerful building block that can be used to avoid sequential bottlenecks in the construction of various spatial acceleration structures. Unlike existing hierarchy generation methods, it outputs nodes in a strict depth-first order and scales well for different workloads as well as for large GPUs. For future work, we believe that our method for constructing octrees could be extended to support triangle meshes through 2.5D rasterization and spatial hashing.

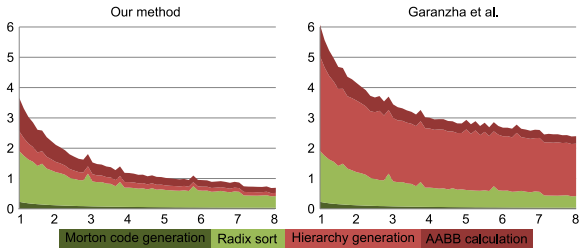

Figure 5: *BVH construction time of Stanford Dragon as a function of the number of cores. Left: The execution time of our method is dominated by sorting the primitives, and scales inversely proportional to the number of cores. Right: The comparison method is dominated by hierarchy generation, which hardly improves even with 8× as many cores.*

Scene	Cores	Common Eval	Common Sort	Common Dup	Our Radix	Our Conv	Prev Seq
Conference Room (283K tris)	1×	0.08	0.78	0.32	0.04	0.24	1.12
	4×	0.03	0.31	0.26	0.03	0.17	0.69
Stanford Dragon (871K tris)	1×	0.22	1.67	0.48	0.46	0.40	1.92
	4×	0.06	0.64	0.31	0.13	0.29	1.45
Turbine Blade (1.77M tris)	1×	0.45	2.73	0.72	0.71	0.50	2.13
	4×	0.12	1.08	0.37	0.20	0.31	1.51

Table 2: *Octree construction times for our method (Our) and Zhou et al. (Prev). Both start by evaluating the Morton codes (Eval), sorting them (Sort), and removing duplicates (Dup). Our method then constructs a radix tree (Radix) and converts it into an octree (Conv), whereas the comparison method performs a sequence of parallel compactions (Seq).*

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics* (2009), pp. 145–149. 1

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum 30*, 7 (2011), 1921–1930. 1

[DPS10] DANILEWSKI P., POPOV S., SLUSALLEK P.: *Binned SAH Kd-Tree Construction on a GPU*. Tech. rep., Saarland University, 6 2010. 1

[Eri04] ERICSON C.: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology)*. 2004. 1

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D. K.: Simpler and faster HLBVH with work queues. In *Proc. High Performance Graphics* (2011), pp. 59–64. 1, 2, 4

[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. 1, 4

[MG10] MERRILL D. G., GRIMSHAW A. S.: Revisiting sorting for GPGPU stream architectures. In *Proc. Parallel Architectures and Compilation Techniques* (2010), pp. 545–546. 4

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

[YB11] YOKOTA R., BARBA L. A.: Fast n-body simulations on GPUs. *ArXiv e-prints* (aug 2011). 1

[ZGHG11] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *IEEE Trans. Vis. Comput. Graph. 17*, 5 (2011), 669–681. 1, 2, 4