

Path Compression Kd-trees with Multi-layer Parallel Construction

A Case Study on Ray Tracing

Zonghui Li*

Yangdong Deng[†]

Ming Gu[‡]

Institute of Software System and Engineering, Software school, Tsinghua University

Abstract

Kd-tree is a fundamental data structure with extensive applications in computer graphics. The performance of many interactive applications such as real-time ray tracing hinges on the construction and traversal efficiency of kd-trees. In recent years, there is a pressing demand for accelerating the construction process due to the fast-growing need of handling dynamic scenes. Existing construction algorithms typically follow a layer-by-layer scheme, which significantly limits the efficiency on the use of multi-core CPUs and GPUs. In this paper, we propose a concurrent multi-layer kd-tree construction algorithm to unleash the inherent parallelism. For a given scene, the algorithm uses Morton code to split its bounding box and orders primitives by Morton curve. A path compression procedure is then concurrently executed on all essential nodes that contain primitives to generate the hierarchy in the target kd-tree. All redundant nodes that have no primitives along the compression paths are collapsed to fast slip empty space. The fully parallel algorithmic scheme adapts variable primitives space and drastically shortens the construction time. A case study on ray tracing benchmarks demonstrates that our kd-tree construction method outperforms the state of art work by an average factor of over 10 and still enables high performance traversal.

Keywords: Kd-tree, Morton curve, Path compression, Multi-layer construction, Ray tracing

Concepts: •Computing methodologies → Ray tracing; Graphics processors;

1 Introduction

As a fundamental data structure, kd-tree has been widely used in computer graphics applications such as K-Nearest-Neighbor computing [Muja and Lowe 2014; Sun and He 2012; Olonetsky and Avidan 2012] high-dimensional filtering [Adams et al. 2009] and ray tracing [Zhou et al. 2008; Choi et al. 2010; Danilewski et al. 2010; Wu et al. 2011]. The efficiency of kd-tree construction algorithms is becoming increasingly more important due to the ever-growing need for interactive computer graphical applications. For instance, the increasing application of ray tracing dynamic scenes poses significant challenges on the construction throughput. Real-time ray tracing applications may need to construct or update and then traverse the underlying kd-tree structure for each frame [Lee et al.

2013; Wald et al. 2007; Parker et al. 2010; Benthin et al. 2012; Keller et al. 2013]. Many researchers proposed excellent solutions to improve the construction efficiency for an interactive frame rate [Zhou et al. 2008; Choi et al. 2010; Danilewski et al. 2010; Wu et al. 2011]. Despite these efforts, the construction process is still the performance bottleneck of ray tracing, especially for large scenes. In addition to construction speed, a high-quality kd-tree usually attempts to maintain a balanced binary tree structure, keep adjacent objects in nearby nodes and allows fast slip of empty space. Existing kd-tree construction algorithms typically build the tree hierarchy in a layer-by-layer fashion to control the tree quality. The problem is that the insufficiently available parallelism at the first a few levels leads to under-utilization of the computing power of modern multi- and many-core processors, especially GPUs. In addition, the obtainable parallelism at a given level can only be known after the preceding level has been processed. Such a fact incurs additional performance overheads such as frequent launching of kernels and load imbalance, which tend to further slowdown the GPU execution.

This paper proposes a fully concurrent multi-layer kd-tree construction algorithm. Our algorithm uses Morton codes to uniformly partition the bounding box that contains all primitives. Such a uniformly splitting corresponds to a fully balanced binary tree, which only exists conceptually and does not really need to be performed. All primitives are encoded with Morton codes and ordered by the Morton Curve. We identify essential characteristics of Morton codes to determine the positions of split planes. A path compression procedure is then launched at all essential nodes that contain primitives. Redundant nodes without primitives are efficiently pruned along the compression path and the final kd-tree structure is concurrently created. Our algorithm computes the whole hierarchy in a fully parallel fashion and completely forgoes the layer-by-layer construction process. Such construction algorithm leads to the following characteristics of our kd-tree.

- **Short Path:** The path compression procedure creates highly compact tree by collapsing all unessential nodes along the paths from essential nodes to the root node.
- **Minimization of Number of Internal Nodes:** In addition, all primitives have the same Morton code are collected into one leaf node and these leaf nodes are ordered by Morton curve. Then every two adjacent nodes generate one internal node. So N leaf nodes only have $N - 1$ internal nodes without singleton nodes like LBVH [Lauterbach et al. 2009]. Such a procedure leads to a minimized number of internal nodes.
- **Fast Pruning of Empty Space:** Empty spaces are collapsed into big ones along with the path compression procedure rather than being collected level-by-level as done in previous kd-tree construction algorithms [Zhou et al. 2008; Choi et al. 2010; Danilewski et al. 2010; Wu et al. 2011]. All leaf nodes then compute their respective bounding box to fast cut the big empty spaces.
- **Adaptive Selection of Split Planes:** Our kd-tree do not alternate split planes in the fixed order of X, Y, and Z as in classical kd-trees. Such a fixed order of alternation is actually an artifact of the layer-by-layer construction. Our path compression

*e-mail: zonghui.lee@gmail.com

[†]e-mail: yangdong.deng@gmail.com

[‡]e-mail: guming@tsinghua.edu.cn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2017 ACM.
I3D '17, February 25-27, 2017, San Francisco, CA, USA
ISBN: 978-1-4503-4886-7/17/03...\$15.00
DOI: <http://dx.doi.org/10.1145/3023368.3023382>

procedure tends to make all essential nodes converge to the center of the primitives and hence the resultant tree adapts to the primitives distribution and keeps a good balance.

All these characteristics above contribute to our high quality kd-tree.

As a case study, we use the proposed method to accelerate kd-tree based ray tracing. In modern ray tracing engines, kd-tree and bounding volume hierarchy (BVH) have been the two most popular acceleration structures. The former [Zhou et al. 2008; Choi et al. 2010; Danilewski et al. 2010; Wu et al. 2011] recursively partitions the space to adapt to the spatial distribution of objects in a top-down manner and combines the well-known Surface Area Heuristic (SAH)[MacDonald and Booth 1990] to improve traversal quality. [Wu et al. 2011] implements a completely SAH-based kd-tree construction on GPUs and presents the state-of-art performance of construction and traversal in kd-tree based ray tracing. The latter[Lauterbach et al. 2009; Pantaleoni and Luebke 2010; Garanzha et al. 2011] follows the same layer-by-layer scheme as kd-tree and clusters primitives by Morton codes. However, [Karras 2012] demonstrates how Morton code enables the maximum parallelism in BVH construction by working on all levels concurrently. This is the state-of-art high performance construction method for BVH. In all of works above, the SAH is still the basic method to achieve high traversal quality. The treelet method [Karras and Aila 2013; Domingues and Pedrini 2015] is appended to the BVH construction and combines with the SAH to improve traversal quality. Although the BVH-based ray tracing illustrates much higher performance than the kd-tree based, this paper selects kd-tree based ray tracing as a case study. Compared with BVH, kd-tree must deal with those split primitives as a performance punishment but does not traverse overlapped nodes as a traversal benefit. So kd-tree used to be considered allowing a higher traversal quality but requiring a longer construction time [Havran 2007]. The performance gap of construction and traversal is more serious and to improve kd-tree construction is more urgent. Fig. 1 illustrates the breakdown of the time for ray tracing typical scenes. In addition, kd-tree is widely used in various applications. The proposed kd-tree construction algorithm is integrated a real time ray tracer. We implement a generic ray-traversal procedure [Horn et al. 2007] to test the tree quality on a set of popular scenes. The experimental results on GPUs demonstrate that the proposed construction algorithm outperforms state-of-the-art results by one order of magnitude in terms of construction speed. Moreover, the construction algorithm allows highly efficient ray traversal process.

Our main contributions are twofold. Firstly, we break through the current layer-by-layer kd-tree construction scheme and propose a concurrent multi-layer construction for kd-tree with full parallelism. Secondly, our method decreases the construction time drastically. As its application, kd-tree based ray tracing is pushed to one new performance level and shortens the performance gap significantly compared with BVH. It also validates the effectiveness of our method.

The paper is structured as follows. In Section 2, we elaborate the algorithmic flow of the proposed kd-tree construction method. Section 3 explains the application of the proposed algorithm under the ray tracing context. Experiment results are also presented. Related works are reviewed in Section 4. Section 5 explains essential implications of the proposed algorithm. We conclude the work in Section 6.

2 Space-adaptive Parallel Construction

The kd-tree construction algorithm is fully parallel by concurrently working on tree nodes. In this section, we elaborate the basic idea

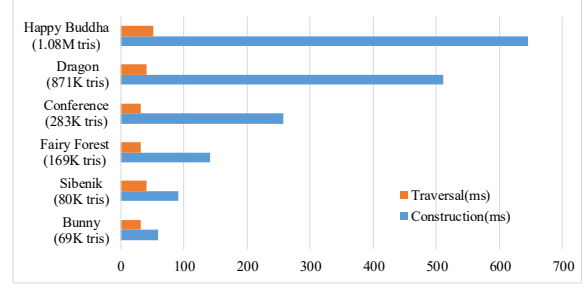


Figure 1: This picture shows the time of construction and traversal kd-trees by a state-of-the-art ray tracer [Wu et al. 2011]. The construction time is significantly longer than traversal, especially in larger scenes. All scenes are from the Stanford 3D Scanning Repository [Stanford].

and the algorithmic details of two key techniques, deriving split planes by Morton code and path compression. The proof of correctness of the proposed method is also presented.

2.1 Deriving Split Planes by Morton Code

Morton code [Morton 1966] is a function based on the well-known space-filling curve, Morton curve or Z-order curve. It discretizes a 3-D space into a regular grid in which each grid is associated with a sub-space. This regular grid naturally corresponds a perfectly balanced, complete binary kd-tree. Fig. 2, *III* gives an example of a balanced tree in the 2-D space. Note that the tree only exists in concept and we use it to illustrate the path compression procedure. If each dimension is discretized with n bits, the Morton code of a given grid consists of $3n$ bits by interleaving the binary digits of 3 dimensions in a 3-D space. The Morton curve defines a linear ordering to preserve the spatial locality. In other words, adjacent Morton codes suggest neighboring positions. Fig. 2, *I* shows an example that one Z-order curve and the corresponding Morton codes for each unit square in a 2-D plane. The primitives of the same Morton code are clustered into the same square. And the Morton codes corresponding to the nearby primitives are also adjacent on the Morton curve. Primitives in a scene naturally fall into the 3-D grid. Hence, each primitive will have its Morton code equal to the corresponding grid. A closer look at the Morton code reveals two essential characteristics for our kd-tree construction algorithm to organize the primitives in tree structure.

Observation 1: Morton code offers a natural way to discover potential split planes according to the positions of primitives. Given two adjacent primitives or sub-spaces and one by one to compare from left to right, the leftmost bit that has a different value in their Morton code is designated as the significant bit because it actually suggests a split plane. The axis corresponding to the significant bit is the direction for splitting. A 2-D example is illustrated in *II*, Fig. 2.

Observation 2: When hierarchically splitting a space along the X, Y, and Z directions, Morton code naturally encodes the parent and child relationship among resultant sub-spaces. Suppose the Morton code of a sub-space has the rightmost bit 1 appearing on the i -th position. The parent sub-space will have the $(i-1)$ -th and i -th bits as 10 in its Morton code. In addition, the corresponding two bits of the left and right child sub-spaces will be 01 and 11, respectively. In Fig. 2, *III* illustrates the observation.

The above observations suggest that Morton code implicitly defines

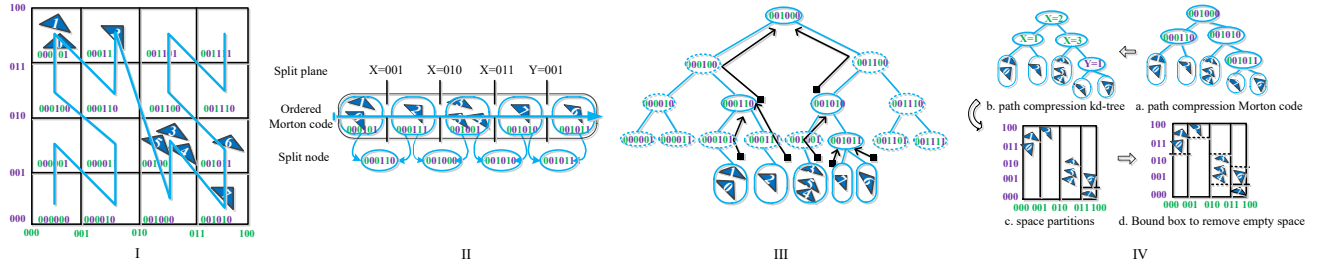


Figure 2: The kd-tree construction process of the proposed algorithm. I: Encoding primitives using Morton code and ordering primitives with the respective Morton curve orders. Green and brown bits indicate X and Y axis, respectively. II: Computing split planes along with Morton curve. III: Applying path compression method on a fully balanced complete tree corresponding to the uniform split by Morton codes in I. Nodes with a solid circle are essential, while the rest nodes redundant. Note that the tree is conceptual and does not need to be actually constructed. IV: Decode KD-tree nodes and space partitions.

a space partition. A kd-tree can be built based on the above characteristics. The Observation 1 can be exploited to find the split planes and The Observation 2 can be utilized to determine the parent-children relationship.

2.2 Path Compression Method

The proposed path compression method allows a fully parallel kd-tree generation process. As illustrated in I, Fig. 2, the path compression method starts with an encoding of all primitives with Morton codes [Morton 1966]. In this paper, we compute Morton codes of triangle primitives by their respective centroid coordinates. Then all Morton codes corresponding to primitives are sorted. Based on the resultant ordering, adjacent Morton code computes the Morton code of the split plane with comparing each other one by one from left to right, illustrated in II, Fig. 2. All Morton codes corresponding to a non-empty set of primitives are leaves nodes in III, Fig. 2, while all internal nodes have a respective split plane associated with a given Morton code. All Morton codes of primitives and split planes are essential nodes and the other nodes are redundant nodes. All essential nodes are marked by solid circles and redundant nodes with dotted circles.

Given a thread as the minimum unit of algorithm execution, the path compression process launches one thread for each essential node. All threads trace their own path toward root node until they reach the next nearest essential ancestor node. In III, Fig. 2, the black arrows are the path that essential nodes trace. If a thread starting from a node reaches an ancestor node from the left(right) side, the node will become the left(right) child of the ancestor in the resultant tree. During path compression process, the redundant nodes along a path are removed. The method concurrently processes essential nodes at all levels and enables completely parallel execution. As a result, the multi-layer hierarchy is generated concurrently. The resultant tree is drawn in IV.a, Fig. 2. Now all nodes are essential nodes including n leaves nodes and $n - 1$ internal nodes that are computed from leaves nodes. The worst case of execution time of a thread is to trace the path from a leaf node all the way up to the root. So the time complexity of the worst case is $O(\log n)$. In real execution, the worst case happens only on very few nodes, which is validated in the Sub-Section 3.2.

The kd-tree exemplified in IV.b, Fig. 2 is generated by manipulating the Morton codes to derive split planes. A special feature of our kd-tree is that the split planes do not alternate in the fixed order of X, Y, and Z as in classical kd-trees. Instead, at each level the split axis adapts to the distribution of primitives in the corresponding space. The space partition of the path compressed kd-tree is

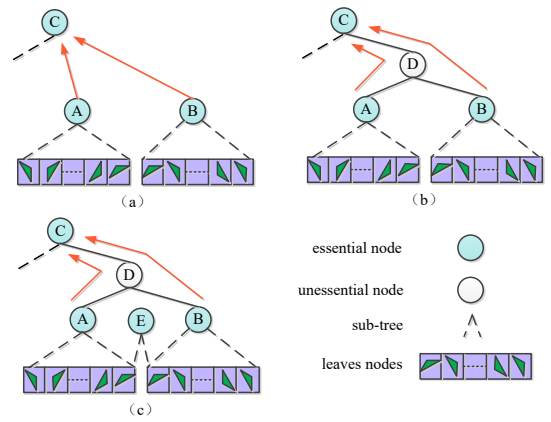


Figure 3: Illustration of correctness proof. (a): Two paths reach node C from the right, i.e. C has two right nodes A and B. (b): Both nodes A and B reach C bypass node D. (c): An essential node E exists between A and B.

depicted in IV.c, Fig. 2. Our path compression procedure uses the least number of split planes to partition the primitives space. Such an strategy collects empty sub-spaces into a big sub-space containing only a few small primitives. The situation can be more obvious on scenes with highly irregular distribution of objects. In Fig. 2, IV.c illustrates such a case. We can compute a bounding box for each leaf node to fast skip large empty spaces. The resultant space partition is presented in IV.d, Fig. 2.

2.3 Proof of Correctness

With our method, the hierarchy generation starts from a complete binary tree conceptually. All threads then perform path compression process independently. To guarantee the correct execution, the tree property must be maintained in the path compression process. Here we present a sketch of our proof on the correctness of the path compression method proposed in this work.

The path compression process begins with performing a collapsing operation upwards in all essential nodes (leaves and split planes nodes). Given a node at a certain level, it will be collapsed by any path arriving from lower levels if it is not essential. If it is an essential node, we have to distinguish two possible situations. First,

the tree property is maintained when two paths arrival from the left and right, respectively. However, the tree structure is destroyed if both paths upwards arrival at the same left or right child nodes. We use proof by contradiction to show that such a situation will not happen.

We assume the situation that two essential nodes A and B go upwards along the path and arrive at node C from the same left or right side of the node C. Without loss of generality, Fig. 3 (a) illustrates the situation from the right side of the node C. As we all know, in tree structure, only one path exists from one node to another node. So before nodes A and B arrive at the node C, they go across the same nodes. We select the first node that both A and B nodes pass by at path compression process as the node D. The situation is represented in Fig. 3 (b). Since D is the first node passed by A and B, A and B belong to the left sub-tree and the right sub-tree of the D, respectively. Without loss of generality, as illustrated in Fig. 3 (b), node A is in the left sub-tree of node D and node B is in the right sub-tree of node D. If the D is one essential node, A and B will stop at node D and keep the tree property. So the D is one unessential node.

On the other hand, the corresponding leaves nodes of the sub-trees A and B are ordered on the Morton curve. The maximum leaf node in sub-tree A and the minimum leaf node in sub-tree B are adjacent, otherwise those leaves nodes between the two leaves nodes will lead that the path compression process of either A or B stops before it reaches node D because those leaves generate essential nodes in the path either from A to D or from B to D. The maximum leaf node in sub-tree A and the minimum leaf node in sub-tree B will generate one node E illustrated in Fig. 3 (c). Since E is the ancestor of both the maximum leaf node in sub-tree A and the minimum leaf node in sub-tree B, E does not belong to neither the sub-tree A nor the sub-tree B. Then E is an ancestor node of both A and B. It then follows that A and B should converge at E from the left and right side of E. So E is the first node passed by both A and B. E is equal to D. This is one contradiction that D is essential and is also unessential. In conclusion, path compression process keeps the tree property during kd-tree construction. Because any two essential nodes will not become the same left child or right child of one ancestor, all threads handling with path compression process are conflict free. Q.E.D.

3 Case Study: Ray Tracing

Enabling photo-realistic visual effects, the ray tracing algorithm is widely used in computer graphics applications. To lower the compute complexity, Kd-trees are one of the most important classes of spatial acceleration structures. A complete ray tracing process for dynamic scenes mainly consists of two stages, construction of the acceleration structure and ray traversal on the structure. As illustrated in Fig. 1, the cost of kd-tree construction can be a serious performance bottleneck. Here we select ray tracing as a case study to demonstrate the advantages of the proposed kd-tree construction method. In the ray tracing oriented implementation, our kd-tree construction algorithm consists of the following steps.

Evaluate Morton Codes: Given the primitives (triangles in this work) in a scene, we first compute the centroid for each triangle and then transform these centroid coordinates into Morton codes and store them in an array.

Copy Triangles: Given planar primitives, we must handle primitives that are cut by split planes. Here we simply copy the triangle into both sub-spaces. The operation will increase the number of triangles. The added triangles will be inserted into the array of Morton codes.

Table 1: The statistics for nodes levels in path compressed kd-trees of various scenes[Stanford]

Scenes&level	Max	Min	Average	St.deviation
Bunny	16	7	14.3	1.5
Sibenik	16	9	14.6	1.2
Fairy	16	10	14.9	1.2
Conference	19	11	16.9	1.4
Dragon	22	8	19.7	1.7
Happy Buddha	22	7	19.8	1.7

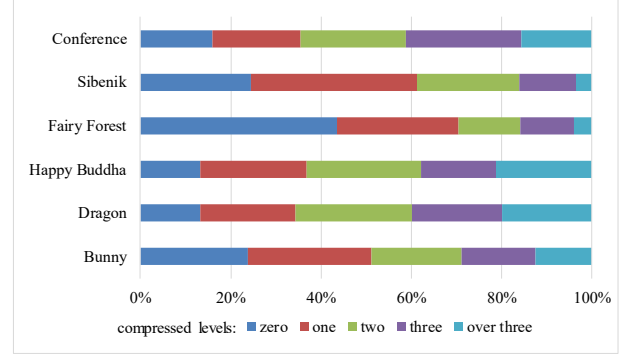


Figure 4: The percentage for nodes compressed different levels in total nodes.

Sort Morton Codes: We use the radix sort in the parallel algorithms library[Thrust 2012] to order these Morton codes.

Generate Leaves Nodes: We cluster the primitives owning the same Morton code into one leaf based on ordered Morton code.

Generate Split Planes Nodes: After generating leaves nodes, each leaf node owns unique Morton code. And these Morton codes are ordered. By comparing adjacent two leaves nodes, we generate $n - 1$ split planes nodes for n leaves nodes. And then, we need to mark these split planes as essential nodes. Here we use one bit for each Morton code. '0' means an unessential node while '1' means an essential node. Finally, we decode these Morton codes for split planes nodes into actual split planes along axis.

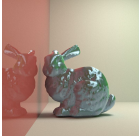



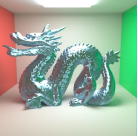

Path Compression Process: Launch one thread for every leaf node and split plane node. All threads go upwards to seek the nearest essential ancestor nodes by collapsing unessential nodes respectively. If it reaches the ancestor node from left side, it becomes the left child of the ancestor node. Otherwise, it becomes the right child of the ancestor node. Path compression process collapses all unessential nodes and generate the hierarchies of kd-tree.

Evaluate Bounding Box per Leaf Node: Combing the small empty spaces resulted from the path compression process into some big spaces. Here we simply evaluate the bounding box for each leaf node to fast skip these big empty spaces.

3.1 Evaluation of the Path Compression Kd-tree

We quantitatively evaluate our path compression based kd-tree construction algorithm for various scenes [Stanford]. Experiment results validate the advantages of our kd-trees.

Table 2: Comparison with kd-tree construction and traversal cost for primary rays only. All reported cost time uses ms as time unit.

						
	Bunny (69K tris)	Sibenik (80K tris)	Fairy Forest (169K tris)	Conference (283K tris)	Dragon (871K tris)	Happy Buddha (1.08M tris)
[Wu et al. 2011] _{construction}	26.8	40.5	71	129	232.2	293.3
ours _{construction}	4.3	4.7	5.6	5.9	8.7	9.3
[Wu et al. 2011]/ours	6.2	8.6	12.6	21.8	26.6	31.5
[Wu et al. 2011] _{traversal}	16.3	20.5	16	15.5	21.5	23.2
ours _{traversal}	8.0	6.2	4.1	8.6	17.8	21.4
[Wu et al. 2011]/ours	2	3.3	3.9	1.8	1.2	1.1

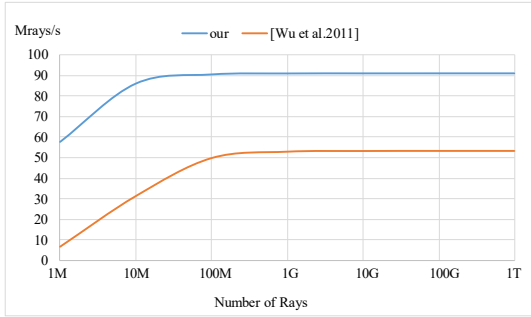


Figure 5: The Ray tracing performance comparison between our kd-tree and the state-of-art kd-tree work [Wu et al. 2011] based on build time and traversal time. The longer slop means to suffer from the long construction time. The higher Mrays per second means one better ray traversal throughput.

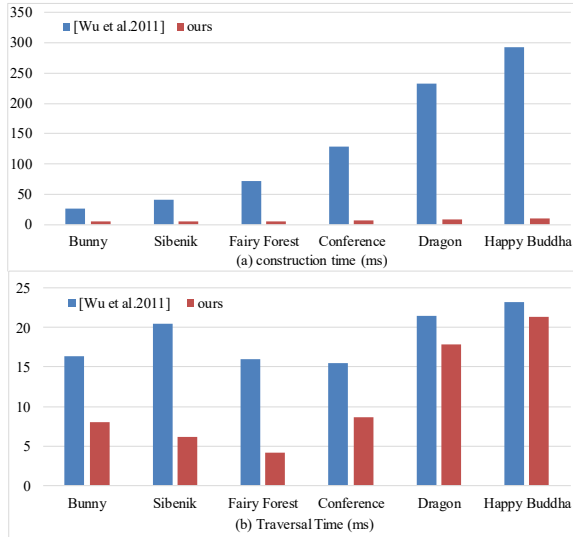


Figure 6: The construction and traversal time comparison between ours and [Wu et al. 2011]. (a) is the comparison of construction time. (b) compares the traversal time.

Good Balance: The path compression method collapses all redundant nodes to generate hierarchies with full parallelism. All essential nodes converge to the center of the primitives space so as to achieve a good balance tree structure. The statistics in Table 1 shows that the average level of nodes is within three levels from the maximum level in the whole kd-tree and the standard deviation keeps less than 2.0. Such statistics demonstrates the resultant trees are rather balanced.

High Efficiency: The worst case of path compression process happens when tracing a path from a leaf node to the root. The time complexity is the $O(\log n)$. As demonstrated in Figure 4, actually the worst case occurs only on very few nodes. In all scenes, the nodes for the zero compressed levels are nearly 20% and the nodes for the not more than two compressed levels almost reach 60%. The nodes for the over three compressed levels are less than 20% except that in the Happy Buddha. The happy Buddha's nodes that whose compressed levels that are not more than three levels also reach 79%. The shorter compressed path means the more computing resource is relaxed to process other tasks.



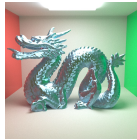

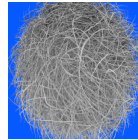

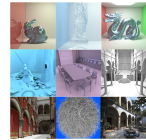
Compactness: The path compression method generates compact kd-trees. It uses a minimized number of internal nodes to construct the whole kd-tree. As a result, it shortens the traversal path from root to other nodes. Moreover, it allows the collection of small empty spaces into some big spaces and simply computes bounding box for leaves nodes to fast slip these empty spaces. In previous kd-tree construction algorithms [Zhou et al. 2008; Choi et al. 2010; Danilewski et al. 2010; Wu et al. 2011], the handling of empty space has to be done level by level.

High Quality: Following the evaluation method proposed by Karras and Aila [Karras and Aila 2013], we quantitatively analyze the ray tracing performance as follows.

$$effective_ray_tracing_performance = \frac{number_of_rays}{rendering_time},$$

where $rendering_time = build_time + \frac{number_of_rays}{ray_throughput}$. We use the average build time and ray throughput of various scenes [Stanford], which are tested on a NVIDIA GTX 580 GPU. Figure 5 illustrates our kd-tree achieves lower construction time and higher ray traversal throughput.

Table 3: Comparison of construction and traversal time with BVH on NVIDIA GTX TITAN for diffuse rays. Diffuse interreflection rays are cast for each intersection point with a random distribution of direction in a hemisphere [Aila and Laine 2009]. KBVH [Karras and Aila 2013], LBVH [Lauterbach et al. 2009], HLBVH [Pantaleoni and Luebke 2010] and SBVH [Stich et al. 2009] are well known BVH builders. The performance of these BVHs is taken from the results of [Karras and Aila 2013] collected on NVIDIA GTX TITAN. The build time unit is ms. The traversal performance unit is MRays per second.

														
	Fairy Forest (169K tris)		Conference (283K tris)		Dragon (871K tris)		Happy Buddha (1.08M tris)		Hairball (2.9M tris)		Samiguel (10.5M tris)		Nine scenes Average	
Builder	Trav. MR/s	Build Time	Trav. MR/s	Build Time	Trav. MR/s	Build Time	Trav. MR/s	Build Time	Trav. MR/s	Build Time	Trav. MR/s	Build Time	Trav. MR/s	Build Time
Our	53.4	4.4	104.4	7.6	26.0	10.5	31.3	16.3	10.2	50.9	25.6	104.8	49.7	23.5
KBVH	305.6	7	275.6	9	213.1	24	181.2	29	50.0	73	83.5	274	192.7	53.6
LBVH	253.6	2	179.4	2	188.5	7	161.4	8	42.0	21	55.3	74	148.8	14.6
HLBVH	253.2	6	185.0	6	189.1	8	160.8	9	38.8	16	39.9	32	147.6	11
SBVH	337.4	3 s	378.4	7 s	237.9	20 s	202.1	25 s	56.4	121 s	118.5	136 s	234.6	40.8 s

3.2 Results and Analysis

The proposed kd-tree construction algorithm was integrated into a GPU based ray tracer for dynamic scenes. We use a relatively simple kd-tree traversal method proposed by [Horn et al. 2007] to testify the tree quality in terms of traversal performance. The nine commonly used test scenes are selected to capture scene-to-scene variation. The CONFERENCE, SIBENIK and CRYTEK-SPONZA scenes are widely used architecture models. The BUNNY, DRAGON and HAPPY-BUDDHA scenes consist of finely tessellated objects. The FAIRY-FOREST scene features widely varying size of triangles. The HAIRBALL scene has a highly complex internal structure. The SANMIGUEL combines architecture with fine geometric detail and vegetation. The complexity of these scenes varies from 69K to 10.5M. All scenes are tested under a resolution of 1024×1024 with both primary rays and diffusion rays. Diffuse rays are generated as reflection rays with the origin at the intersection points of primary rays and the direction following a random distribution in the hemisphere [Aila and Laine 2009]. The performance measurements are computed as the averages over six viewpoints to reduce the dependence of viewpoint. We evaluate the performance under the contexts of both kd-tree and BVH.

Kd-tree Comparison: Our ray tracer was tested on a NVIDIA GTX 580 graphics card (Fermi architecture) since it is close architecture to GTX 280 (Tesla architecture) that [Wu et al. 2011] report his results based on. Hence, we translated the computation times by a performance divisor, which is derived by measuring the performance gap between GTX 280 with GTX 580 on three SmallLuxGPU benchmarks [SmallLuxGPU 2011]. These benchmarks have their computing patterns and workloads similar to tree construction and traversal. The results conversion ratio of ray traversal was also confirmed by benchmarking the architecture independent while-while kernel [Aila and Laine 2009] with similar workloads. Finally, the performance divisor for tree construction was also validated by comparing relevant parallel primitives on GPUs with different architectures. We select the same six commonly used scenes [Stanford] rendered by our ray tracer as those in [Wu et al. 2011], which is illustrated in table 2. To the best of our knowledge,

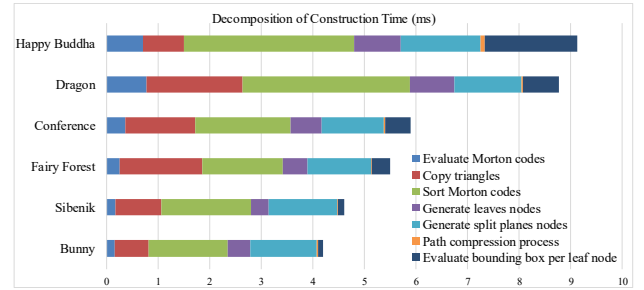


Figure 7: The cost distribution of different time steps in kd-tree construction.

[Wu et al. 2011] reported the best kd-tree construction results for ray tracing. We compare our results with that in [Wu et al. 2011]. The comparison for kd-tree construction and traversal is reported in Table 2. As demonstrated in Table 2, the Kd-tree construction time of our algorithm is impressive due to dramatically shorten construction cost. On these commonly used scenes reported by [Wu et al. 2011], our construction time is 5 to 30 times shorter. Moreover the bigger the scenes are, the more obvious the decreased time. Figure 6(a) demonstrates the trends. Kd-tree always serves special destinations in various applications. Here kd-tree construction is used for ray traversal of fast intersection test. As illustrated in Table 2, our kd-tree delivers high efficiency traversal and shorten their traversal time for all these various scenes. The conclusion is demonstrated intuitively in Figure 6(b).

To further reveal the details of our kd-tree construction algorithm, we depict the distribution of construction time among major processing steps on various benchmark scenes in Figure 7. Sorting Morton code is the most consuming time step. We note that copying triangles also cost much time. However, the operation is not one general step because planar primitives will be split into sub-parts by split planes. In point clouds, the copy operation does not exist. The path compression process costs very little time. It demonstrates the

hierarchy generation by path compression process is very high performance.

BVH Comparison: To illustrate the shortened gap between BVH and kd-tree based ray tracing, we also evaluate our ray tracer on NVIDIA GTX TITAN and compare with the state-of-art BVH works. The results are illustrated in Table 3. They clearly show that our method has pushed the kd-tree construction performance to a similar or even higher level of that of BVH. For traversal performance, the diffuse rays throughput(Mrays/s) is demonstrated in Table 3. Although our kd-tree presents one lower traversal, our kd-tree construction is one straightforward construction without the well-known SAH [MacDonald and Booth 1990] and the treelet [Karras and Aila 2013][Domingues and Pedrini 2015]. All BVHs in Table 3 use SAH to improve traversal quality. [Karras and Aila 2013] combines the treelet and SAH to achieve high quality traversal and such BVH we call as KBVH. The SAH and treelet could also be applied to our kd-tree for higher quality in future work. In addition, our traversal algorithm[Horn et al. 2007] is a relatively simple algorithm since our work targets on kd-tree construction. The Kepler-Dynamic-Fetch optimized kernel [Aila et al. 2012] could be used in our traversal algorithm in future work to achieve high traversal performance, which has been implemented in all these BVHs traversal in Table 3. The straightforward construction and simple traversal fully demonstrate the potential of kd-tree for higher quality traversal. Moreover the traversal results also demonstrate the shortened performance gap between BVH and kd-tree based ray tracing. Our algorithm allows comparable traversal performance with leading BVH traversal algorithms.

4 Related work

The idea of building spatial data structures with Morton codes is not new. It has been widely used for BVH construction. Especially, a few recent works have achieved superior performance by following such a heuristic [Lauterbach et al. 2009; Pantaleoni and Luebke, 2010; Garanzha et al. 2011; Karras 2012]. We also use Morton code in this work but the novelty of our approach is that we bring in the path compression idea for multi-layer construction by the inherent characteristics of Morton codes.

Linear Bounding Volume Hierarchy (LBVH) [Lauterbach et al. 2009] is constructed by first ordering all primitives along a Morton curve and then recursively splitting the sequence to create nodes. Hierarchical LBVH (HLBVH) [Pantaleoni and Luebke 2010] is an extension of LBVH by improving the coherence among threads and global memory behaviors. [Garanzha et al. 2011] proposed a further enhancement by introducing working queues to improve parallelism. Both LBVH and HLBVH are performed in a top-down, level-by-level manner with limited parallelism at upper levels. Based on a conceptual complete binary tree, our construction algorithm does not need to create new nodes at each level and thus enables a fully parallel construction process. In addition, LBVH-alike data structures tend to have singleton nodes or even singleton chains of nodes, which introduces unnecessary overhead during traversal [Lauterbach et al. 2009]. Our path compression technique completely removes such singleton nodes. [Karras 2012] introduced a fully parallel construction method for BVH. It declared the adaption for kd-tree but no implementation was reported. For a given essential node, [Karras 2012] starts from an upper bound l_{max} and performs a binary search in $[0, l_{max}]$ to find the other end of the range. This is typically downwards sub-tree search to probe child nodes. It provides another way to implement our path compression process. Compared with our up-forwards compression for the nearest essential parent node, it searches for child nodes along the path.

Table 4: The performance comparison between the automatically set bit count and the adjusted bit count for each scene. The bit count value is per dimension and we assume that the number of bits for each dimension is equal. The build time unit is m.s. The traversal performance unit is MRays per second.

Scenes	Automatic Value			Adjusted Value		
	Bit Cnt	Build Time	Trav. MR/s	Bit Cnt	Build Time	Trav. MR/s
Bunny	4	3.0	42.6	5	3.3	67.5
Sibenik	4	3.3	38.3	5	3.6	51.1
Fairy	5	4.4	53.4	5	4.4	53.4
Conference	5	5.6	86.9	6	7.6	104.4
Dragon	5	8.6	14.3	6	10.5	30.0
Buddha	6	11.9	23.3	7	16.3	31.3
Sponza	5	6.5	67.9	6	10.2	77.9
Hairball	6	50.9	10.2	6	50.9	10.2
Sanmiguel	7	80.1	23.0	8	104.8	25.6

5 Discussion

In our kd-tree construction, Morton code is used to split the bounding box of a scene and generates the final split planes in targeted kd-tree. The initial bit number of Morton code impacts the performance of our kd-tree certainly. Firstly we automatically decide the bit count for each scene. We simply assume that the scene is uniform distribution and each cell space corresponding to one Morton code contains 32 triangles since one warp has 32 threads in GPUs. Based on the assumption, we achieve one original bit count for each scene. And then from the start value, we manually adjust the bit count value to minimize the total of the construction time, the primary rays trace and the second rays (diffuse rays) trace time. Table 4 shows that the performance comparison of automatic bit count and the adjusted bit count. Compared with the automatic values, the adjusted values usually has more one bit. The more one bit brings the construction punishment since more bits split scene space into smaller space cell as a result that more split triangles need to be handled and more triangles are copied. But traversal benefit is achieved due to less big nodes that has many triangles. However, no results suggest that the more bit count, the better traversal performance. In fact, when the bit count increases persistently, the copy punishment will replace the traversal benefit fast because of highly increasing triangles duplication. Such punishment is also the essential reason that BVH construction is usually faster than kd-tree construction. To achieve the best bit count is complex. It is related to the distribution of primitives, the geometrical shape of primitives such as planar or point, the strategy to deal with split primitives. Fortunately the simple evaluation for bit count like ours could also achieve one not bad performance result, illustrated in Table 4. Meanwhile, one more accurate evaluation for bit count certainly improves our kd-tree performance.

6 Conclusions

In this work, we propose a fully parallel multi-layer construction algorithm for spatial acceleration structures to overcome the limitation of traditional layer-by-layer construction methods. We identify inherent characteristics of Morton code to directly determine split planes. A path compression procedure is then developed to remove unessential nodes and generate the final tree structure. The full parallelism characteristic makes it especially suitable for parallel com-

puting. When applied to the ray tracing application, the proposed algorithm dramatically shortens construction time and still enables high traversal efficiency. The proposed algorithm offers new opportunities to improve the performance of many important applications in graphics and other domains.

Acknowledgements

Anonymous reviewers for comments. Samuli Laine for HAIR-BALL, Guillermo Leal Laguno for SANMIGUEL, Stanford Computer Graphics Laboratory for BUNNY, SIBENIK, FAIRY FOREST, CONFERENCE, DRAGON, HAPPY BUDDHA.

This research is partially supported by NSFC Program (Grants No.91218302 and No.61527812), National Science and Technology Major Project (No.2016ZX01038101, MIIT IT funds (Research and application of TCN key technologies) of China, and The National Key Technology R&D Program (No.2015BAG14B01-02).

References

- ADAMS, A., GELFAND, N., DOLSON, J., AND LEVOY, M. 2009. Gaussian kd-trees for fast high-dimensional filtering. In *SIGGRAPH*.
- AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of High-Performance Graphics*, 145–149.
- AILA, T., LAINE, S., AND KARRAS, T., 2012. Understanding the efficiency of ray traversal on gpus - kepler and fermi addendum. in NVIDIA Technical Report NVR-2012-002.
- BENTHIN, C., WALD, I., WOOP, S., ERNST, M., AND MARK, W. R. 2012. Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture. *IEEE Transactions on Visualization and Computer Graphics* 18, 9, 1438–1448.
- CHOI, B., KOMURAVELLI, R., LU, V., SUNG, H., BOCCHINO, R., ADVE, S., AND HART, J. 2010. Parallel sah kd-tree construction. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 77–86.
- DANILEWSKI, P., POPOV, S., AND SLUSALLEK, P., 2010. The frobnicatable foo filter. Technical report, Saarland University.
- DOMINGUES, L. R., AND PEDRINI, H. 2015. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, 13–20.
- GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster hlbvh with work queues. In *ACM Siggraph/eurographics Conference on High PERFORMANCE Graphics 2011, Vancouver, Canada, August*, 59–64.
- HAVRAN, V. 2007. About the relation between spatial subdivisions and object hierarchies used in ray tracing. In *Spring Conference on Computer Graphics*, 43–48.
- HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive kd tree gpu raytracing. In *Proceedings of ACM Symposium on Interactive 3D graphics and games*, 167–174.
- KARRAS, T., AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *High-Performance Graphics Conference*, 89–99.
- KARRAS, T. 2012. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, 33–37.
- KELLER, A., KARRAS, T., WALD, I., AILA, T., AND LAINE, S. 2013. Ray tracing is the future and ever will be... In *Proceedings of SIGGRAPH Courses*.
- LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. *Computer Graphics Forum* 28, 2, 375–384.
- LEE, W., SHIN, Y., LEE, J., LEE, S., RYU, S., AND KIM, J. 2013. Real-time ray tracing on future mobile computing platform. In *SIGGRAPH Asia Symposium on Mobile Graphics and Interactive Applications*.
- MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput.* 6, 3 (May), 153–166.
- MORTON, G. M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, Ottawa, Canada: IBM Ltd.
- MUJA, M., AND LOWE, D. G. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11, 2227–2240.
- OLONETSKY, I., AND AVIDAN, S. 2012. Treecann - k-d tree coherence approximate nearest neighbor algorithm. In *ECCV*, 602–615.
- PANTALEONI, J., AND LUEBKE, D. 2010. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *ACM Siggraph/eurographics Conference on High PERFORMANCE Graphics 2010, Saarbrücken, Germany, June*, 87–95.
- PARKER, S., BIGLER, J., DIETRICH, A., FRIEDRICH, H., AND HOBEROCK, J. 2010. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics* 29, 4, 157–166.
- SMALLLUXGPU, 2011. Smallluxgpu. <http://www.luxrender.net/wiki/SLG>.
- STANFORD. The stanford 3d scanning repository. <https://graphics.stanford.edu/data/3Dscanrep>.
- STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 7–13.
- SUN, J., AND HE, K. 2012. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *CVPR*, 111–118.
- THRUST, 2012. Thrust v1.6.0. <http://code.google.com/p/thrust/>.
- WALD, I., MARK, W. R., GUNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the art in ray tracing animated scenes. *Computer Graphics Forum* 28, 6, 1691–1722.
- WU, Z., ZHAO, F., AND LIU, X. 2011. Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 71–78.
- ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *Proceedings of SIGGRAPH Asia*, vol. 27, 1–11.