DOI: 10.1111/cgf.12776

COMPUTER GRAPHICS *forum*
Volume 35 (2016), number 8 pp. 68–79

Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing

Marek Vinkler[1], Vlastimil Havran[2] and Jiří Bittner[2]

[1]Faculty of Informatics, Masaryk University, Czech Republic
xvinkl@fi.muni.cz
[2]Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic
havran@fel.cvut.cz, bittner@fel.cvut.cz

Abstract
We present a performance comparison of bounding volume hierarchies and kd-trees for ray tracing on many-core architectures (GPUs). The comparison is focused on rendering times and traversal characteristics on the GPU using data structures that were optimized for very high performance of tracing rays. To achieve low rendering times, we extensively examine the constants used in termination criteria for the two data structures. We show that for a contemporary GPU architecture (NVIDIA Kepler) bounding volume hierarchies have higher ray tracing performance than kd-trees for simple and moderately complex scenes. On the other hand, kd-trees have higher performance for complex scenes, in particular for those with high depth complexity. Finally, we analyse the causes of the performance discrepancies using the profiling characteristics of the ray tracing kernels.

Keywords: ray tracing, performance comparison, object-partitioning, space-partitioning, GPU

ACM CCS: I.3.7 [Computational Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

1. Introduction

Solving visibility by ray tracing stands at the core of a number of rendering algorithms, particularly those aiming at computing global illumination. The efficiency of the ray tracing algorithm has significant influence on the total rendering time, and thus much research work has been devoted to ray tracing optimization. The main factors influencing the ray tracing performance are the properties and the organization of data structures which spatially index the scene geometry with the aim to reduce the number of operations needed to find ray-primitive intersections. Over the past decades, two acceleration data structures became prominent for this task: bounding volume hierarchies (BVHs) and kd-trees. These two data structures have been compared against each other a few times on different computer architectures. Due to the development of parallel many-core architectures and more optimized algorithms, most of these studies are now outdated.

In this paper, we compare the ray tracing performance of these acceleration data structures irrespective of their build times, therefore targeting offline rendering algorithms and/or static scenes. For these applications, the time of building the data structure is insignificant compared to the rendering time.

2. Related Work

Below we present shortly the most relevant work related to the data structures tested and evaluated in our paper and the performance studies comparing the two data structures.

BVHs The BVHs [WHG84] have been recently studied in the context of ray traversal efficiency and build algorithms on the GPU. Kopta *et al.* [KIS*12] focused on fast updates during animation while keeping high traversal efficiency. Bittner *et al.* [BHH13] and Karras and Aila [KA13] both target improvement of quality of the already built BVHs. The method of Bittner *et al.* runs on a CPU and produces highest quality BVH, while the method of Karras and Aila runs on a GPU and produces slightly lower quality trees, but in significantly less time. The factors influencing the BVH traversal times on many-core processors were analysed by Aila *et al.* [AKL13], and Guthe [Gut14] gave some reasons to understand the discrepancy between the surface area heuristic (SAH) cost [GS87] and the measured performance. Gu *et al.* [GHFB13] presented a method for building BVHs using agglomerative clustering that allows for setting a trade-off between build time and traversal efficiency.

Kd-trees The kd-trees introduced to ray tracing by Kaplan [Kap85] have been recently studied in the context of parallelization of the building algorithm on both CPUs [CKL*10], [RPC12] and GPUs [DPS10], [WZL11], [RPC12]. Choi *et al.* [CKL*10] focused on accelerating SAH kd-tree build algorithm on multi-core CPUs. To simplify the parallelization, they omitted split clipping [HB02] from the method, which resulted in lower quality trees. Danilewski *et al.* [DPS10] presented a scalable build algorithm with binning for building SAH kd-trees on GPUs. Wu *et al.* [WZL11] moved the entire kd-tree build algorithm of Wald and Havran [WH06] including split clipping to the GPU, significantly accelerating the algorithm. A hybrid CPU–GPU implementation of the same baseline algorithm was proposed by Roccia *et al.* [RPC12], that outperformed the previous approaches.

Hybrid hierarchies Wächter and Keller [WK06] proposed a data structure Bounding Interval Hierarchy (BIH) that uses two splitting planes per node. In BIH, all primitives lie completely in either the left or right child and thus the costly duplication of primitives straddling the splitting plane is prevented. Stich *et al.* [SFD09] presented another hybrid data structure Spatial Bounding Volume Hierarchy (SBVH) that keeps the hierarchy of bounding volumes as in BVH, but allows for splitting primitives into smaller ones as in kd-tree. This prevents large overlaps of bounding volumes and thus increases the rendering performance.

Performance studies The efficiency of acceleration data structures for ray tracing was compared in several studies. Havran [Hav00] formulated hardware-independent measures and studied properties of 12 acceleration data structures. He concluded at that time (year 2000), that kd-trees have the highest ray traversal performance on average from all the data structures tested for static scenes on CPUs. Another comparison study by Masso and Lopez [ML03] showed similarly to Havran [Hav00] that BVHs built up by insertion using the algorithm of Goldsmith and Salmon [GS87] are significantly worse than top–down built kd-trees. On the GPUs, data structures for ray tracing were compared by Zlatuška and Havran [ZH10]. Their study targeted older GPU hardware (GTX 280 and 8600GT) of year 2007/2008 and showed that the BVHs are the fastest for coherent rays, while the kd-trees are the fastest for incoherent rays.

The utility of dynamic data structures proposed for animated scenes were surveyed by Wald *et al.* [WMG*09]. In the survey, the authors suggest that kd-trees are not efficient for dynamic scenes because of their slow rebuild and propose to use uniform grids or BVHs depending on the distribution of primitives in the scene. This survey, however, lacks a direct quantitative performance comparison of the discussed methods.

3. Data Structures

We build the two data structures, a binary BVH and a binary kd-tree, fully on the GPU using the algorithm of Vinkler *et al.* [VBHH13]

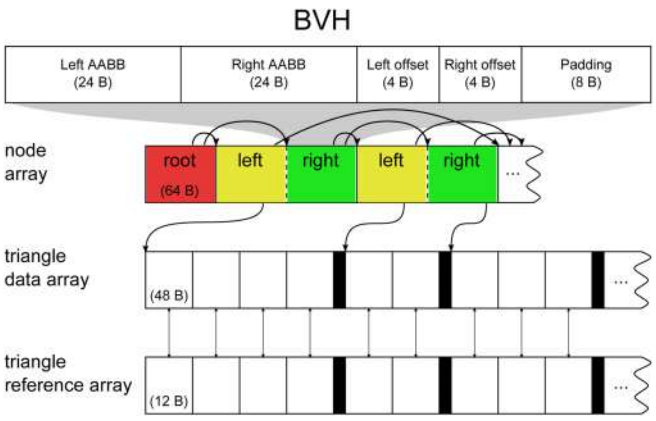

Figure 1: *Memory and node layout of BVHs used in our measurements. For BVHs, the triangle data array is accessed directly by the traversal algorithm. The leaf is terminated with a stop mark (black rectangle in the figure). The BVH node takes 64 bytes: 24 bytes for each of the children axis-aligned bounding boxes (AABB), two 4 byte offsets to the left and right children and an 8 byte padding.*

which is integrated with the framework of Aila and Laine [AL09]. We have further extended this framework with GPU code for traversing kd-trees. The kd-tree traversal kernel is organized similarly to the BVH traversal kernel in the while-while layout [AL09] and with the same triangle intersection test. We also build the hybrid SBVH on the CPU for comparison.

3.1. BVHs

Build algorithm We build the BVHs with the SAH used for subdividing inner nodes and for automatic termination of the subdivision. The SAH cost is evaluated in each node for a fixed set of candidate planes (binning). The best splitting plane according to SAH is chosen from 32 splitting plane candidates uniformly distributed inside the bounding box of a node in all three axes (11 planes parallel to the x-axis, 11 planes parallel to the y-axis and 10 planes parallel to the z-axis). If all 32 candidate planes fail to subdivide the geometric primitives into two non-empty parts, the primitives are subdivided into two equally sized parts. The fixed number of candidate planes is chosen to fit the warp size on the current generation of GPUs, thus allowing for efficient implementation of the build algorithm.

A leaf is created when (1) the number of triangles in a node is smaller than or equal to a given threshold ($n_{\max} = 4$, triangle count criterion) or (2) when the SAH cost of not subdividing a node is less than the cost of subdividing it (cost criterion). The value $n_{\max} = 4$ was chosen from values 2, 4, 8 and 16 as it gives the fastest traversal times for shooting incoherent rays over our test scenes (see Table 1). The depth of the hierarchy is not limited since the build is always terminated by the two criteria.

Memory layout Figure 1 shows the memory layout of our BVHs, as proposed by Aila and Laine [AL09]. The size of each node in the node array is 64 bytes with children of each node stored in a consecutive chunk of memory. The most significant bit of the

Table 1: *The dependence of the ray tracing performance of the BVH and the kd-tree on the maximum number of triangles in a leaf (n_{max} is 2, 4, 8 and 16). $P_{primary}$ is the ray tracing performance for primary rays in Mrays s^{-1}, and $P_{diffuse}$ is the ray tracing performance in Mrays s^{-1} for shooting eight diffuse sample rays per primary ray. The average performance is computed from the average traversal time in milliseconds. Values in boldface highlight the best performing parameters.*

	BVH								Kd-tree							
Scene	P_{primary}[Mrays s^{-1}]				P_{diffuse}[Mrays s^{-1}]				P_{primary}[Mrays s^{-1}]				P_{diffuse}[Mrays s^{-1}]			
$n_{\max}$	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
Sibenik	**263**	**263**	256	217	**71**	69	61	46	**222**	**222**	217	182	**40**	**40**	37	30
Fairy Forest	**170**	**170**	**170**	156	**69**	**69**	66	54	119	**121**	114	101	**40**	39	36	29
Crytek Sponza	**164**	**164**	156	141	**47**	46	41	33	**154**	152	145	127	**38**	**38**	34	26
Conference	**255**	**255**	251	232	**73**	72	71	62	197	**198**	195	174	**50**	49	45	36
Powerplant16	147	**148**	147	142	**73**	**73**	**73**	68	**113**	112	109	92	**57**	56	52	40
Dragon	278	**282**	264	223	**183**	182	167	135	127	128	**129**	113	**86**	84	79	61
Happy Buddha	238	**244**	233	200	175	**177**	161	129	122	124	**125**	111	**87**	86	79	64
Office House	**66**	**66**	**66**	63	**19**	**19**	**19**	18	**135**	133	128	114	**42**	41	39	34
Sodahall	**313**	**313**	294	263	**117**	115	107	89	**278**	263	233	196	**87**	84	75	59
Hairball	57	58	**61**	**61**	34	34	**35**	34	**34**	**34**	33	29	**23**	**23**	22	18
Houses 3×3	**182**	**182**	170	147	**96**	95	86	69	152	**154**	145	116	**82**	80	72	55
Asian Dragon	105	**107**	104	91	136	**138**	130	109	**76**	75	73	61	**84**	82	75	60
San Miguel	62	**63**	**63**	58	**22**	**22**	21	18	**60**	59	55	46	**20**	19	17	14
MPII subset	**141**	139	137	130	**57**	**57**	55	50	**192**	189	175	145	**83**	80	73	55
Houses 6×5	127	**130**	127	111	64	**65**	61	51	135	**137**	125	105	**71**	69	61	48
Powerplant	62	62	**63**	**63**	19	19	**20**	19	**103**	101	94	75	**37**	36	33	26
Four SPD [Hai87] scenes with triangles used in [Hav00]																
mount8	**526**	**526**	476	400	**396**	381	332	268	154	156	**159**	141	**104**	**104**	99	79
sombrero4	**417**	385	345	286	**328**	320	286	233	213	217	**233**	204	**163**	162	156	131
teapot40	**455**	**455**	435	370	**286**	280	248	202	**164**	**164**	161	143	**104**	103	93	76
tetra8	**667**	**667**	556	500	**606**	**606**	479	417	130	**132**	124	108	**160**	159	141	113
Average	142	**143**	141	130	**59**	**59**	57	50	**115**	**115**	110	95	**53**	52	48	38

child offset determines whether the offset points to an inner node or the first triangle in a leaf. Triangles falling to the same leaf are stored sequentially in memory and followed by a stop mark (0x80000000) that specifies the end of the leaf. Triangle's vertex data in Woop's representation [Woo04] are stored in the triangle data array and take up $3 \times 16 = 48$ bytes. Triangle references are stored in a separate triangle reference array with each reference occupying $3 \times 4 = 12$ bytes. Even if only 4 bytes are sufficient for the reference index, the data are padded to 12 bytes so that the same offset can be used to access both the triangle data and the triangle reference arrays. The triangle reference array is used to identify the hit triangles indices which are used later, e.g. for shading computation. The triangle array layout supports a single triangle to be stored in multiple leaves through duplication of the triangle's data but this is not used in our BVHs.

Traversal algorithm For tracing the rays, we use the speculative while-while traversal algorithm of Aila and Laine [AL09] which proved to be the fastest one for the BVHs. This traversal algorithm is stack-based and stores the traversal stack in local memory, which is part of the GPU DRAM. The algorithm traces the rays separately of each other (not using packet traversal algorithm) leading to high performance on incoherent rays. The speculative traversal allows for higher utilization of the parallel cores and in turn higher performance.

3.2. Kd-trees

Build algorithm Similarly to BVHs, we build SAH kd-trees using 32 splitting plane candidates in each node evenly distributed along the x, y, z extents of its axis-aligned bounding box. Given the axis-aligned bounding box of an interior node, the splitting plane candidates are distributed the same way as for the BVH. Unlike for the BVHs, when all triangles fall into one child of a node for the candidate split with the minimal cost, an empty leaf is created for the other child.

The kd-trees are built with split clipping [HB02] to get high-quality trees. With split clipping enabled a triangle is only considered to fall into a child if it intersects its bounding box. In the original sequential algorithm, the bounding boxes of all triangles and their fragments are maintained, and shrunk upon intersections by the splitting planes. These auxiliary bounding boxes are then used for more precise triangle-child intersection tests. This technique is difficult to implement on GPUs as it requires frequent dynamic memory allocation for the newly created auxiliary bounding boxes. We implement split clipping by directly computing the intersection of each triangle with the bounding boxes of the children of the currently subdivided node. The new formulation that repeatedly computes the intersections between triangles and bounding boxes does not require to keep the auxiliary boxes, but is more computationally demanding. The proposed solution fits the GPU architecture well. We use the algorithm of Akenine-Möller [AM05] for the intersection of an axis-aligned box and a triangle.

We use the termination criteria of Havran and Bittner [HB02] with modified constants to achieve fast ray tracing. On GPUs, the termination criteria also influence the divergence of the ray-primitive intersection tests, making selection of the constants more complex. A kd-tree leaf is created when (1) the number of triangles in a node is smaller than or equal to $n_{\max} = 2$ (triangle count criterion) or (2) the depth of the node is higher than the maximum allowed depth, $d_{\max} = k_1 \cdot \log_2 N + k_2$ with $k_1 = 1.2$ and $k_2 = 2.0$ (node depth criterion) or (3) the node has failed to pass the cost criterion several times. The cost criterion specifies failure in terms of the cost of a subdivided node and an unsubdivided one: $C_{\text{new}}/C > r_q^{\min}$ with $r_q^{\min} = 0.9$. The maximum number of failures of the cost criterion along the path from the root to the current node is set to $F_{\max} = K^1_{\text{fail}} + K^2_{\text{fail}} \cdot d_{\max}$ with $K^1_{\text{fail}} = 1.0$ and $K^2_{\text{fail}} = 0.26$.

The value of $n_{\max}$ was chosen from values 2, 4, 8, 16 as it gives the fastest ray traversal times over our test scenes (see Table 1). The value of k_1 was selected from three values $\{1.0, 1.1, 1.2\}$ and k_2 was selected from the range $\langle 2.0, 8.0\rangle$ with step 1.0. The value of $r_q^{\min}$ was selected from the range $\langle 0.7, 1.2\rangle$ with step 0.1 to get the highest performance as shown in Figure 2. Similarly, K^2_{fail} was selected from the range $\langle 0.15, 0.35\rangle$ with step 0.01 to provide overall the highest performance for the ray traversal algorithm. For sufficiently high $r_q^{\min}$ the values of constants K^1_{fail} and K^2_{fail} do not matter since the subdivision of the node is always considered as successful. The constants k_1, k_2, K^1_{fail} and K^2_{fail} are used for tuning the build process based on the scene complexity.

Memory layout Figure 3 shows the memory layout of our kd-tree, which is inspired by the BVH layout, but it is more memory efficient when triangles are split into multiple references. The children of an inner node are again stored in a consecutive chunk of memory with each node taking 16 bytes. Similar to the BVH, the most significant bit of the child offset differentiates between an inner node offset and a triangle offset. A triangle offset with the largest negative value is reserved for empty leaves so that no extra memory is required for them. The flag in the node layout holds information about the axis of the split plane. The triangle offset points to the triangle reference array instead of the triangle data array to prevent duplication of triangle data. Only triangle references are duplicated and each reference occupies just 4 bytes to save space. The triangle references falling into a leaf are again stored in consecutive memory and the leaf is terminated with a stop mark as in the BVH layout.

Traversal algorithm The ray traversal algorithm is similar to the one for the BVH, but its speculative variant is not used. We have experimented with this optimization and found out it slows down the traversal algorithm when used for kd-trees. We suspect this is caused by the more scattered memory access of kd-tree representation leading to increased memory bandwidth and latency. Moreover, this optimization mitigates the benefit of the early termination and increases code complexity. According to the classification used in [HH11], we are using a recursive stack-based traversal algorithm with the near/far sorting of child nodes based on ray direction (see Algorithm 1). The traversal stack keeps just two values per entry (node address and exit distance) instead of the three (node address, entry distance and exit distance).

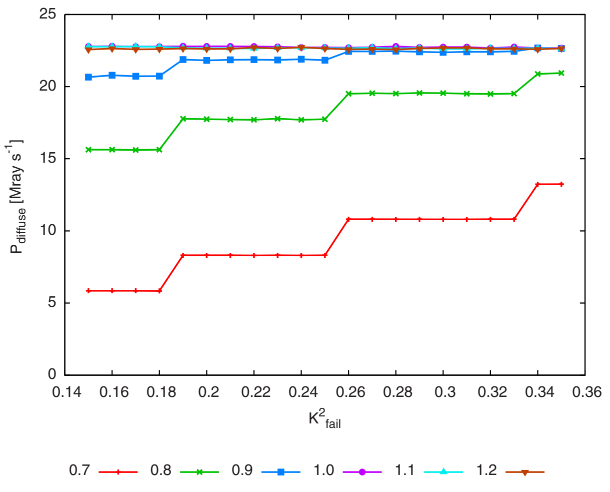

Figure 2: *Dependence of the ray tracing performance for kd-trees on the r_q^{min} and K^2_{fail} constants, averaged over all scenes.*

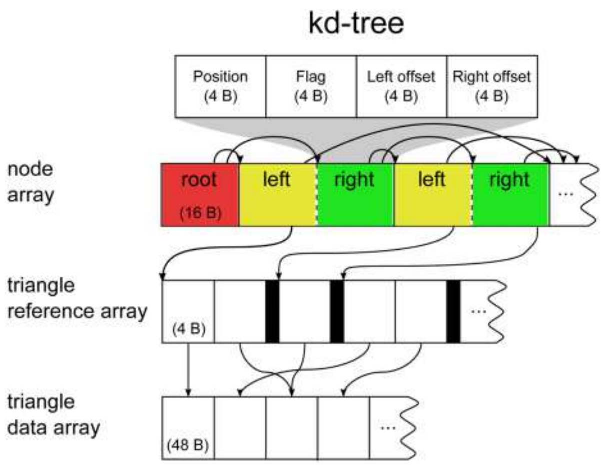

Figure 3: *Memory and node layout of kd-trees used in our measurements. For kd-trees, the triangle data array is accessed with a dependent memory access through the triangle reference array because the triangle references may be duplicated. The leaf is terminated with a stop mark (black rectangle in the figure). The kd-tree node takes 16 bytes: the position of the splitting plane stored in 4 byte floating point value, a 4 byte information for the type of the node and two 4 byte offsets to the child nodes.*

Algorithm Traversal code for the kd-tree, *tmin* is the entry distance of a ray in the current node and *tmax* is the exit distance in the current node

```
Traverse() begin
    Intersect ray with scene AABB, compute tmin and tmax;
    while (not hit and tmax > tmin) do
        while (node is inner node and tmax > tmin) do
            Fetch Kdtree node;
            t ← (split − orig) / dir;
            Choose near/far based on ray direction;
            if (t ≥ tmax) then
                node ← near;
            else if (t ≤ tmin) then
                node ← far;
            else
                node ← near;
                push(far, tmax);
                tmax ← t;
        while (node is leaf) do
            for (each triangle in leaf) do
                intersect triangle, store distance in t;
                tmax ← t;
            if (not hit) then
                tmin ← tmax;
                pop(node, tmax);
            else
                break;
```

3.3. Hybrid data structures

Build algorithm For building the SBVH [SFD09], we are using the CPU-based implementation provided in the framework of Aila and Laine [AL09]. This is a high-quality SAH builder that chooses the best splitting plane using both the object partitioning and space partitioning strategies. For object partitioning, the sweep-based evaluation is used (exact SAH evaluation), while for space partitioning 32 bins are used in each axis. The best object partitioning splitting plane is then compared to the best space partitioning splitting plane to decide which to use to subdivide the node. The termination criteria are set the same as for the BVH build, i.e. $n_{\max} = 4$ and SAH cost termination.

Memory layout The same memory layout as for the BVH is used since it already allows for triangle duplication in multiple leaves. The lengths of the triangle reference array and triangle data array, however, no longer correspond to the number of input triangles.

Traversal algorithm Since SBVH shares the same memory layout with BVH, it can also use the same traversal algorithm. Thus, we are again using the speculative while-while traversal algorithm.

4. Results

We evaluated the algorithms on a PC with Intel Core i7-2600, 16 GB of RAM and NVIDIA GeForce GTX 680 running 64-bit Windows 7.

4.1. BVHs versus kd-trees

Both data structures were tested on 20 scenes with a varying number of primitives and depth complexity. Four of the scenes were taken from the Standard Procedural Database (SPD) [Hai87] to allow comparison on the triangular scenes with the older performance study of Havran [Hav00]. The measured data were averaged over four viewpoints for the non-SPD scenes, while for the SPD scenes a single viewpoint given by the SPD proposal was used. The images of rendered scenes are shown in Figure 4. The ray tracing performance was measured five times for each viewpoint and the maximum performance is reported.

The summary results from measurements with BVHs and kd-trees on our test scenes are shown in Table 2. Since the kd-tree is a space subdivision data structure, it contains more nodes and triangle references (pointers to the same triangle data) than the BVH for the same input scene. The number of triangle references cannot be estimated in advance. In our measurements up to 14.6× more nodes and 11.8× more triangle references were created for kd-trees than for BVHs for the Hairball scene. This increases both the build times and the memory footprint of kd-trees.

Multiple reference to triangles due to splitting in kd-trees can both increase or decrease the ray tracing performance. The decrease of the ray tracing performance happens because more memory accesses are required, and those are more scattered in the address space. On the other hand, the capability to decrease the triangle bounds and the early termination of the ray traversal algorithm upon the first hit allows the kd-tree to achieve lower numbers of ray-triangle intersection tests (N_{it}) and traversal steps (N_{ts}). The reduction in the number of algorithmic steps can become significant for complex scenes with high depth complexity. For example, in the Office House scene, the kd-tree performs only 17% ray-triangle intersection tests (N_{it}) and 30% traversal steps (N_{ts}) compared to the BVH and this reduction leads to a two-fold ray tracing speedup. For some other scenes, even significant reduction in the number of operations need not translate to faster ray tracing, such as for the San Miguel scene. When the number of algorithmic steps (both traversal steps and intersection tests) is roughly similar (Happy Buddha and teapot40) for both BVHs and kd-trees, then the ray tracing with kd-trees is about twice slower than for the BVHs.

Differentiating performance by the scene complexities, the BVH is usually more efficient on small to medium-sized scenes and less efficient on large scenes as shown in the graph in Figure 5. The Office House scene is an exception to this rule (small to medium-sized scenes) because it is formed by many large triangles causing significant overlaps of BVH nodes, so the ray tracing with kd-tree is about twice faster than with BVH.

Even if the build times for the data structures on the GPU are not the subject of our study, we mention them briefly. We have found out that for top–down build algorithms the times are by factor of

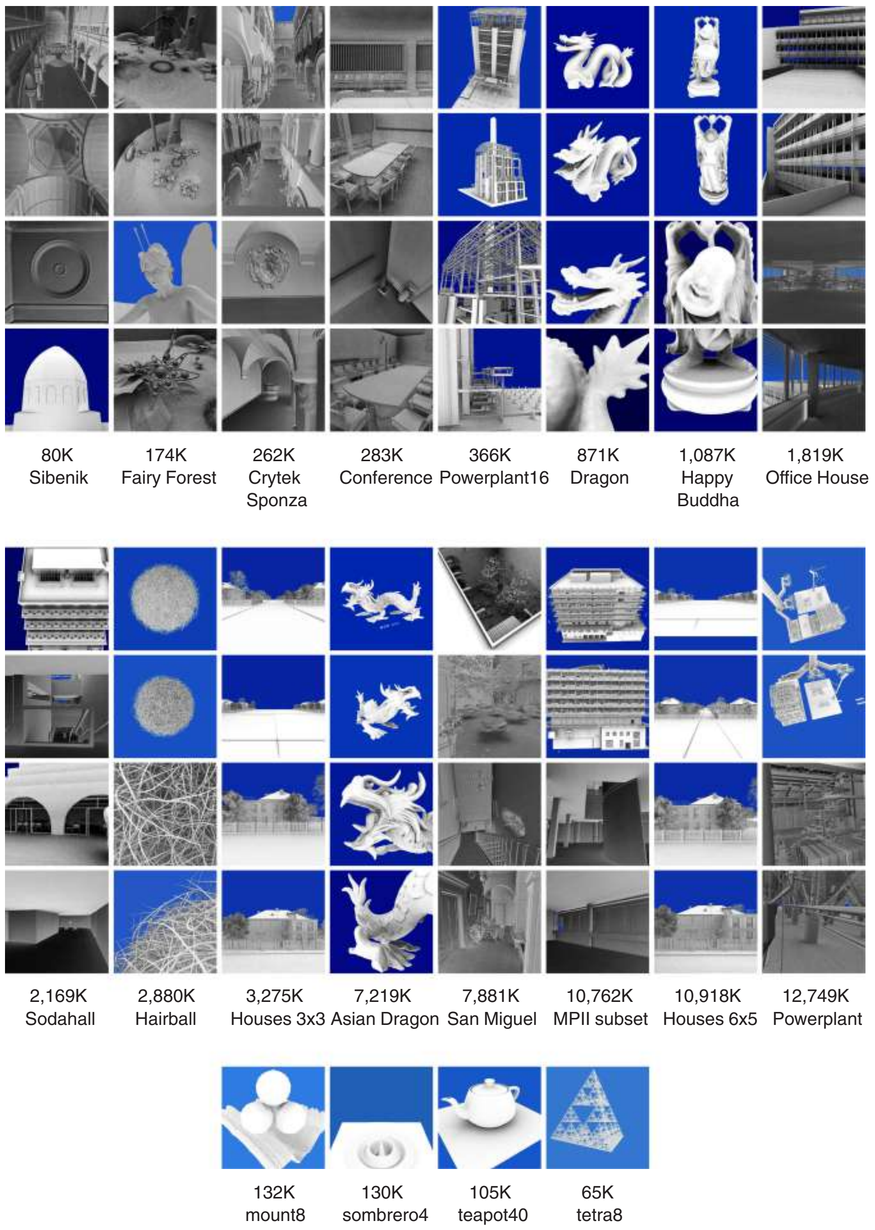

Figure 4: *Rendered images of our 20 test scenes in resolution 1024 × 1024 pixels. Eight diffuse samples per primary ray were used for rendering. For the scenes from the Standard Procedural Database (SPD) [Hai87] only the viewpoints defined by this database were used. For the other scenes, four representative viewpoints were chosen. The MPII subset consists of layers #1,4,7,8,9,10,12,13 of the whole model [HZDS09].*

10 to 20 times higher for kd-trees than for BVHs. This is due to the need for the dynamic memory allocation required for kd-trees that is rather slow on a GPU, even if we use an optimized memory allocator for GPUs [VH14]. Moreover, the triangle splitting also increases the number of triangles that must be repeatedly sorted in the lower levels of the tree, thus, increasing the memory traffic. The use of split clipping in kd-tree build algorithm decreases the rendering times, but usually at the cost of increasing the build times. When not using the split clipping, the build times for kd-trees are decreased by approximately 20%.

In our measurements, we have tested four SPD scenes [Hai87] consisting purely of triangles (as our software framework supports only such scenes) that were also used in Havran's PhD thesis [Hav00]. Comparing the results from our kd-tree implementation to those in Havran's work, measured for the same geometry data and the same viewpoints, we can see that similar trees are built (number of nodes, triangle references and traversal characteristics). The main difference in the build algorithm is that we use only 32 uniformly distributed candidate splitting planes instead of all the reasonable planes. Using only a few planes was reported by several authors to

Table 2: *Comparison of various statistics of BVHs and kd-trees (KDT) on 20 scenes including ratios of some characteristics. N_{tris} is the number of scene triangles, N_G/N_{tris} is the number of built nodes divided by the number of scene triangles, N_{ref}/N_{tris} is the number of references to triangles divided by the number of scene triangles (1.0 for BVHs because there is no triangle splitting), $Memory$ is the summed memory consumed by the nodes and triangle references of the data structure in Mbytes (MB), N_{it} is the number of ray-triangle intersection tests per ray, N_{ts} is the number of traversal steps per ray, $P_{primary}$ is the ray tracing performance for primary rays in Mrays s^{-1} and $P_{diffuse}$ is the ray tracing performance for eight diffuse samples in Mrays s^{-1}. The average performance is computed from the average traversal time in milliseconds. The columns denoted as 'ratio' give the ratio from numbers of the two previous columns on the left. In particular, the value of the performance ratio column indicates the speedup of the kd-tree over the BVH (value greater than one corresponds to the kd-tree being faster than the BVH). Values in boldface highlight the data structure that performed better.*

	N_{tris}	N_G/N_{tris} [−]			$N_{\text{ref}}/N_{\text{tris}}$ [−]		$Memory$ [MB]		$N_{\text{it}}(primary)$ [−]			$N_{\text{ts}}(primary)$ [−]			P_{primary} [Mrays s^{-1}]			P_{diffuse} [Mrays s^{-1}]		
Scene	[−]	BVH	KDT	ratio	BVH	KDT	BVH	KDT	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio
Sibenik	80K	**0.32**	0.87	2.73	**1.0**	3.4	**1.87**	2.11	8.6	**8.1**	0.94	64.4	**36.8**	0.57	**263**	222	0.84	**70**	40	0.58
Fairy Forest	174K	**0.31**	0.91	2.91	**1.0**	3.5	**3.99**	4.75	12.4	**11.2**	0.90	64.2	**51.0**	0.79	**169**	120	0.71	**69**	40	0.57
Crytek Sponza	262K	**0.30**	1.36	4.49	**1.0**	4.2	**5.84**	9.72	12.4	**7.6**	0.61	95.7	**49.0**	0.51	**164**	154	0.94	**46**	38	0.83
Conference	283K	**0.32**	0.71	2.20	**1.0**	3.4	**6.66**	6.75	**9.6**	9.9	1.04	51.3	**28.6**	0.56	**254**	197	0.78	**72**	50	0.70
Powerplant16	366K	**0.29**	1.17	4.04	**1.0**	5.6	**7.87**	14.4	12.8	**6.8**	0.53	60.8	**37.8**	0.62	**147**	113	0.77	**73**	57	0.78
Dragon	871K	**0.33**	1.92	5.81	**1.0**	4.6	**20.9**	40.9	**3.2**	3.9	1.21	31.0	**29.5**	0.95	**282**	127	0.45	**182**	85	0.47
Happy Buddha	1,087K	**0.33**	2.32	7.02	**1.0**	5.0	**26.1**	59.4	**3.9**	4.0	1.03	35.0	**29.6**	0.85	**244**	122	0.50	**177**	86	0.49
Office House	1,819K	**0.27**	0.65	2.38	**1.0**	3.3	**36.0**	39.7	58.8	**9.8**	0.17	142.7	**42.6**	0.30	66	**132**	2.00	19	**42**	2.17
Sodahall	2,169K	**0.30**	1.03	3.43	**1.0**	3.8	**48.1**	66.1	5.6	**4.5**	0.80	64.1	**41.3**	0.64	**313**	278	0.89	**115**	87	0.75
Hairball	2,880K	**0.30**	4.42	14.63	**1.0**	11.8	**64.1**	325	24.8	**16.7**	0.67	101.3	**77.3**	0.76	**57**	34	0.60	**34**	23	0.68
Houses 3×3	3,275K	**0.33**	1.84	5.66	**1.0**	4.6	**77.5**	151	5.9	**4.4**	0.75	45.1	**31.6**	0.70	**182**	154	0.85	**94**	82	0.87
Asian Dragon	7,219K	**0.32**	1.77	5.56	**1.0**	4.3	**168**	315	4.8	**3.6**	0.74	**35.2**	37.3	1.06	**107**	76	0.71	**138**	84	0.61
San Miguel	7,881K	**0.31**	1.19	3.82	**1.0**	3.8	**181**	259	16.5	**6.6**	0.40	135.9	**77.5**	0.57	**63**	60	0.94	**22**	20	0.89
MPII subset	10,762K	**0.28**	1.28	4.56	**1.0**	4.5	**226**	397	9.3	**3.4**	0.37	83.3	**34.8**	0.42	139	**213**	1.53	57	**91**	1.61
Houses 6×5	10,918K	**0.32**	2.60	8.00	**1.0**	5.4	**258**	661	8.2	**4.5**	0.55	55.3	**35.3**	0.64	128	**147**	1.15	65	**75**	1.16
Powerplant	12,749K	**0.26**	0.95	3.70	**1.0**	3.9	**248**	378	46.3	**8.5**	0.18	114.9	**49.7**	0.43	62	**112**	1.81	19	**40**	2.08
Four SPD [Hai87] scenes with triangles used in [Hav00]																				
mount8	132K	**0.28**	1.75	6.33	**1.0**	4.4	**2.72**	5.72	**5.2**	5.9	1.13	**30.8**	33.9	1.10	**385**	156	0.41	**248**	106	0.43
sombrero4	130K	**0.29**	1.78	6.24	**1.0**	4.5	**2.76**	5.81	**2.6**	3.5	1.35	18.3	**16.5**	0.90	**526**	213	0.40	**386**	164	0.42
teapot40	105K	**0.30**	2.12	7.03	**1.0**	5.2	**2.34**	5.52	**3.3**	3.4	1.03	**26.6**	26.7	1.00	**370**	167	0.45	**225**	106	0.47
tetra8	65K	**0.25**	2.34	9.36	**1.0**	5.5	**1.25**	3.73	**1.7**	3.5	2.06	**16.7**	23.1	1.38	**588**	137	0.23	**541**	163	0.30
Average	–	**0.30**	1.65	5.49	**1.0**	4.7	**69.4**	137.5	12.8	**6.5**	0.82	63.6	**39.5**	0.74	**142**	117	0.85	**59**	54	0.84

only slightly decrease the quality of the data structure [HKRS02], [HMS06].

Given the built kd-trees are only slightly lower quality than the ones of Havran [Hav00] we can compare the ray traversal performances. The measurements for primary rays show that only the development of the hardware accounts for an average performance speedup of 1070× for tracing rays in a span of 15 years. Interestingly, this practically matches the progress predicted by Moore's law (doubling of computation performance every 18 months): speedup of $2^{10} = 1024\times$ for 15 years (single core Pentium II in 1997 to massively parallel GPU architecture Kepler GK104 in 2012). This is most likely thanks to ray tracing being efficiently parallelizable. This conclusion cannot be generalized to other algorithms or the CPU to GPU comparison [Lee10].

Our setup for testing the data structures for ray tracing significantly differs from the one of Zlatuška and Havran [ZH10] in several aspects. We measure our results on the NVIDIA Kepler architecture introduced in 2012, that is two generations newer than the NVIDIA Tesla architecture used in the paper by Zlatuška and Havran [ZH10] (year 2007/2008, GeForce 6 Series and GeForce 200 Series). In particular, hardware caches were introduced in the newer hardware, making stack-based ray traversal methods more efficient. We use the single ray traversal algorithm [AL09] for BVHs, while their study used a packet traversal one [GPSS07]. The kd-tree ray traversal algorithms differ as well: we used a modified algorithm of Aila and Laine [AL09], while Zlatuška and Havran used the algorithm of Horn *et al.* [HSHH07] based on short stack with infrequent restarts of the ray traversal from the root node. In particular, the packet ray traversal algorithm used in the older study caused the BVHs to be faster only on coherent rays, while they were significantly slower for the incoherent ones. The traversal algorithms for BVHs and kd-trees compared in our study are much more similar, showing the fundamental differences between the two acceleration data structures. Moreover, only scenes up to 1 million primitives (Happy Buddha) were used in the original paper, while we compare scenes up to 12 million primitives (Powerplant) where the kd-trees provide better performance than the BVHs as shown in Table 2. Last but not least, the older study used data structures built up on a CPU and then

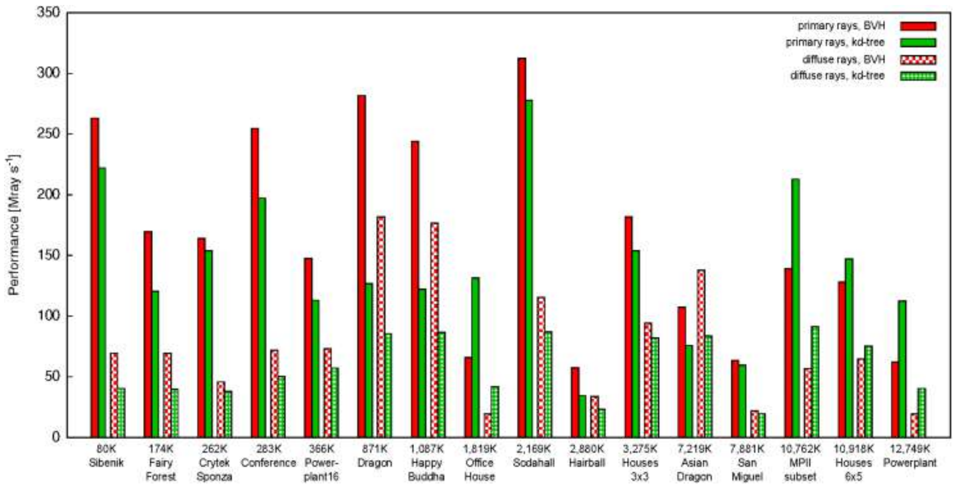

Figure 5: *Performance of BVHs and kd-trees on our non-SPD test scenes for both primary (solid colour) and diffuse rays (colour pattern). The scenes are sorted in ascending order with the number of triangles increasing to the right. For very large scenes, the kd-tree performs faster than the BVH.*

Table 3: *Comparison of the performance of SBVHs, BVHs and kd-trees (KDT) on 16 scenes. $P_{primary}$ is the ray tracing performance for primary rays in Mrays s^{-1}, and $P_{diffuse}$ is the ray tracing performance for eight diffuse samples in Mrays s^{-1}. The average performance is computed from the average traversal time in milliseconds. For SBVH, the measured performance is given, while for BVH and kd-tree, the values indicate the speedup over the SBVH (values lower than one correspond to the SBVH being faster). Values in boldface highlight the data structure that performed the best.*

	N_{tris}	P_{primary} [Mrays s^{-1}]			P_{diffuse} [Mrays s^{-1}]		
Scene	[−]	SBVH	BVH	KDT	SBVH	BVH	KDT
Sibenik	80K	**327**	0.81	0.68	**97**	0.72	0.42
Fairy Forest	174K	**240**	0.71	0.50	**98**	0.71	0.40
Crytek Sponza	262K	**206**	0.80	0.75	**72**	0.63	0.53
Conference	283K	**308**	0.83	0.64	**106**	0.67	0.47
Powerplant16	366K	**301**	0.49	0.38	**162**	0.45	0.35
Dragon	871K	**337**	0.84	0.38	**215**	0.85	0.40
Happy Buddha	1087K	**293**	0.83	0.42	**209**	0.84	0.41
Office House	1819K	**226**	0.29	0.58	**71**	0.27	0.59
Sodahall	2169K	**368**	0.85	0.76	**159**	0.73	0.55
Hairball	2880K	**61**	0.94	0.56	**41**	0.83	0.57
Houses 3×3	3275K	**258**	0.70	0.60	**146**	0.65	0.56
Asian Dragon	7219K	**183**	0.59	0.41	**213**	0.65	0.39
San Miguel	7881K	**123**	0.52	0.48	**45**	0.49	0.43
MPII subset	10 762K	**304**	0.46	0.70	**156**	0.37	0.59
Houses 6×5	10 918K	**246**	0.52	0.60	**138**	0.47	0.55
Powerplant	12 749K	**197**	0.31	0.57	**79**	0.25	0.51
Average	–	**204**	0.60	0.54	**97**	0.50	0.48

ray tracing of these data structures on a GPU. However, we are able to build up both data structures on a GPU directly, using relatively similar algorithms for BVHs and kd-trees.

4.2. Hybrid data structures

We have also tested a variant of BVH called SBVH [SFD09] that allows to reduce the size of the primitive bounding boxes by spatial splitting them into two references by the means of the splitting plane. This method represents a hybrid concept between the BVH and the kd-tree.

A parallel build algorithm of SBVHs for GPUs has not been proposed yet as it likely faces more challenges than the construction of kd-trees, especially with increased size of the data structure and temporary storage which would likely not fit into the main memory on current GPUs for large scenes. Thus, for our comparison, we are using the highest quality CPU builder. Note that the parallel build algorithm of Karras and Aila [KA13] implements an algorithm in the spirit of Ernst and Greiner [EG07] and not of the SBVH as described by Stich *et al.* [SFD09] and Popov *et al.* [PGDS09].

The performance of SBVH is compared to the one of BVH and kd-tree in Table 3. The hybrid algorithm is always faster than the other two because it combines the lower numbers of ray-triangle intersection tests (N_{it}) and traversal steps (N_{ts}) of the kd-tree with the benefit of better mapping to the GPU hardware of the BVH.

We have not implemented and measured BIH [WK06] since it was shown to have lower ray tracing performance than BVH [Wal07].

5. Limitations

First, our study is limited to scenes composed of triangles only. For primitives with a higher intersection cost, the benefit of the kd-tree doing less intersection tests than the BVH may be more profound. Secondly, we have used and studied stack-based algorithms that are reported to be faster on the GPU than the stackless ones [ASK14].

Thirdly, in our comparison we apply binning with 32 uniformly distributed candidate splitting planes for the selection of the best splitting plane for both BVHs and kd-trees. This choice may lead

Table 4: *Selected profiling characteristics of BVHs and kd-trees (KDT) on 20 scenes including ratios of the characteristics. F_{ops} is the number of performed floating point operations in millions, $DBranch$ is the percentage of divergent branches, $LoadG$ is the amount of memory loaded from GPU DRAM in MB, $StoreL$ is the amount of memory stored to local memory (part of GPU DRAM) majority of which is consumed by traversal stack traffic, $LoadL$ is the amount of memory loaded from local memory (part of GPU DRAM) majority of which is consumed by traversal stack traffic and $\mathrm{L1}_{hit}$ is the percentage of $\mathrm{L1}$ hits for these loads. The value of the ratio column indicates the ratio of the kd-tree over the BVH. The last row holds the average values over the tabled data. Values in boldface highlight the data structure that performed the best.*

	$F_{\mathrm{ops}}[-]$			$DBranch[\%]$			$LoadG[MB]$			$StoreL[MB]$			$LoadL[MB]$			$\mathrm{L1}_{\mathrm{hit}}[\%]$		
Scene	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio	BVH	KDT	ratio
Sibenik	1067	**310**	0.29	**3.7**	6.8	1.85	**335**	750	2.24	**48**	131	2.73	**69**	77	1.11	**94**	78	0.83
Fairy Forest	1171	**391**	0.33	**8.4**	13.7	1.63	**722**	1446	2.00	**99**	266	2.70	**145**	179	1.24	**91**	69	0.76
Crytek Sponza	1557	**303**	0.19	**5.2**	10.0	1.93	**516**	1081	2.10	**109**	226	2.07	161	**131**	0.81	**93**	77	0.83
Conference	924	**293**	0.32	**4.5**	9.7	2.17	**426**	789	1.85	**43**	139	3.23	**62**	91	1.46	**96**	81	0.85
Powerplant16	1154	**268**	0.23	**8.2**	14.0	1.71	**721**	1315	1.82	**92**	138	1.50	**139**	178	1.28	**91**	71	0.78
Dragon	495	**178**	0.36	**12.2**	21.8	1.78	**441**	1370	3.10	**60**	245	4.12	**109**	155	1.41	**92**	58	0.64
Happy Buddha	564	**179**	0.32	**13.3**	23.9	1.80	**512**	1414	2.76	**73**	334	4.59	**132**	171	1.30	**90**	55	0.61
Office House	2348	**342**	0.15	**4.5**	12.1	2.69	1779	**1240**	0.70	**155**	234	1.51	190	**145**	0.76	**89**	73	0.83
Sodahall	974	**219**	0.22	**2.3**	5.7	2.52	**226**	608	2.69	**43**	134	3.08	**57**	74	1.31	**86**	75	0.87
Hairball	2044	**550**	0.27	**11.1**	20.8	1.87	**2038**	4529	2.22	**311**	1088	3.50	**423**	737	1.74	**88**	66	0.75
Houses 3×3	746	**198**	0.26	**10.4**	18.6	1.79	**591**	1117	1.89	**95**	239	2.51	**149**	163	1.09	**85**	64	0.75
Asian Dragon	587	**183**	0.31	**14.9**	25.1	1.68	**1209**	2148	1.78	**131**	481	3.67	**217**	241	1.11	**77**	47	0.61
San Miguel	2196	**356**	0.16	**9.5**	19.5	2.04	**1585**	2849	1.80	**303**	744	2.46	**432**	438	1.01	**84**	65	0.76
MPII subset	1321	**193**	0.15	**6.3**	12.5	1.98	**597**	794	1.33	**93**	216	2.32	143	**108**	0.76	**89**	66	0.74
Houses 6×5	937	**199**	0.21	**10.8**	18.5	1.72	**874**	1130	1.29	**128**	247	1.92	190	**166**	0.87	**81**	64	0.78
Powerplant	1643	**315**	0.19	**9.2**	15.5	1.68	2250	**1416**	0.63	**231**	345	1.49	301	**219**	0.73	**84**	68	0.81
Four SPD [Hai87] scenes with triangles used in [Hav00]																		
mount8	550	**239**	0.43	**7.9**	19.0	2.41	**348**	1208	3.47	**41**	278	6.85	**68**	157	2.32	**94**	64	0.68
sombrero4	318	**146**	0.46	**8.1**	22.9	2.83	**303**	946	3.12	**21**	190	9.04	**37**	79	2.12	**96**	57	0.59
teapot40	448	**174**	0.39	**8.6**	20.6	2.40	**327**	1096	3.35	**36**	240	6.68	**65**	135	2.06	**92**	67	0.73
tetra8	263	**137**	0.52	**9.3**	25.8	2.77	**226**	1362	6.03	**24**	344	14.17	**44**	246	5.62	**95**	61	0.65
Average	1065	**259**	0.29	**8.4**	16.8	2.06	**801**	1430	2.31	**107**	313	4.01	**157**	195	1.51	**89**	66	0.74

to data structures with slightly lower quality than testing all the planes coinciding with the bounding boxes of triangles (maximum $6N$ candidate splitting planes for N triangles for all three axes). Binning has been extensively studied for data structure build and using only a few planes was reported to decrease the quality of the data structure just slightly. For kd-trees, Hurley *et al.* [HKRS02] showed that there is little benefit for having more than 32 split plane candidates per axis and using just eight split planes per axis results in a slowdown of less than 10%. Hunt *et al.* [HMS06] used eight uniform and eight adaptive planes per axis and concluded that the increase in rendering time was less than 4%. For BVHs, Wald [Wal07] used seven candidate planes per axis with testing just the longest axis. The rendering performance was kept within 9% of the exact sweep build algorithm. Günther *et al.* demonstrated that the BVHs build algorithm with binning can sometimes outperform the sweep build one even when using a few splitting planes [GPSS07].

We have also evaluated our own builder when using 64 splitting plane candidates on our test scenes. According to expectations, this leads to a maximum speedup of 10% compared to using 32 planes and using more planes actually decreased performance on two of the scenes. The choice of the number of the splitting planes thus does not influence the conclusions drawn from our comparison of BVHs and kd-trees since the measured differences are higher than the differences made by using more planes.

6. Discussion

We have implemented and optimized the data structures and the corresponding ray traversal algorithms to utilize the features of the GPU architecture, which has a different data processing workflow compared to the CPU (cache-based vs. stream-based model). We will discuss these issues below and document them by the numbers from measurements.

6.1. BVHs versus kd-trees

We have analysed our ray traversal kernels on primary rays using the NVIDIA Nsight version 2.2 [NVI12]. The detailed profiling characteristics measured on all our test scenes are shown in Table 4.

Below we use for the discussion the Happy Buddha scene, for which the number of intersection tests and traversal steps is similar for both the BVH and the kd-tree, so that the other measured characteristics can be meaningfully compared. The number of executed floating point operations, is clearly in favour of the kd-tree (179 million operations for kd-tree vs. 564 million operations for

Table 5: *Selected profiling characteristics of SBVHs, given as an ratio of SBVH values to reference BVH values on 20 scenes. The legend is the same as in Table 4.*

Scene	F_{ops} [−] ratio	$DBranch$ [%] ratio	$LoadG$ [MB] ratio	$StoreL$ [MB] ratio	$LoadL$ [MB] ratio	$L1_{hit}$ [%] ratio
Sibenik	0.80	0.84	0.74	0.76	0.74	1.01
Fairy Forest	0.79	0.81	0.56	0.67	0.70	1.03
Crytek Sponza	0.82	0.79	0.66	0.69	0.68	1.02
Conference	0.89	0.92	0.84	0.88	0.88	0.92
Powerplant16	0.52	0.82	0.40	0.27	0.54	1.02
Dragon	0.92	0.90	0.81	0.62	0.80	1.01
Happy Buddha	0.91	0.89	0.80	0.78	0.77	1.01
Office House	0.43	1.24	0.23	0.43	0.51	1.05
Sodahall	0.83	0.79	0.81	0.78	0.77	1.07
Hairball	0.84	1.01	0.77	1.16	1.12	1.01
Houses 3×3	0.89	0.82	0.59	0.67	0.69	1.05
Asian Dragon	0.83	0.89	0.59	0.65	0.68	1.09
San Miguel	0.72	0.73	0.41	0.51	0.51	1.12
MPII subset	0.59	0.71	0.46	0.47	0.47	1.04
Houses 6×5	0.74	0.77	0.40	0.50	0.56	1.09
Powerplant	0.61	0.80	0.20	0.34	0.38	1.08
Average	0.73	0.88	0.60	0.63	0.66	1.03

BVH). This is expected since the cost of a single traversal step is higher for the BVH and the ray traversal algorithm is carried out speculatively. However, GPUs have high performance in floating point operations and the difference in executed operations does not significantly influence the comparison.

Furthermore, the percentage of divergent branching (percentage of branches where different threads of a warp took different paths and the warp's execution must be serialized, to the total number of branches) is worse for the kd-tree which does not use the speculative traversal (13.3% for BVH with speculative ray traversal algorithm vs. 23.9% for kd-tree). We have also profiled a modified BVH traversal kernel without speculative traversal and verified that it is slower than the speculative version and its percentage of divergent branching is 18.3%.

While Aila and Laine [AL09] report that the traversal kernel is compute limited for the BVH, no such study has been conducted for the kd-trees. Our analysis below shows that the kd-tree ray traversal algorithm is memory limited (bandwidth and latency), explaining the results of Table 2.

There are significant differences between the memory access patterns of both data structures. For the BVH, the two children's bounding boxes are fetched with coherent memory access, while for the kd-tree several accesses are needed to fetch a similar amount of geometrical information. Reading several nodes for a kd-tree instead of just one for the BVH, that represent roughly the same geometric information, increases the amount of transferred data from memory (512 MB for BVH vs. 1414 MB for kd-tree) because of the unused data in each memory fetch. Although this can be mitigated by an improved memory layout of the kd-tree, the two children 64 byte node of the BVH proposed in [AL09] would be likely difficult to conquer. Furthermore, the latency for accessing the same amount of triangle geometry is higher for kd-trees as well due to the dependent memory fetches to GPU DRAM. The memory latency has been recently identified as a bottleneck of the ray traversal performance for the BVHs as well [Gut14], but in our opinion it has even higher impact on the results for the kd-tree.

The traversal kernel for kd-trees requires to save two items on the traversal stack (node address and traversed distance along a ray, in total 8 bytes), while the BVH ray traversal algorithm saves only the node address (4 bytes). When different threads in a warp write into different stack indices the memory writes are not coalesced and the data transfer further increases. For the Happy Buddha scene, this accounts for an increased size of the data stored to local memory (73 MB for BVH vs. 334 MB for kd-tree). Given the traversal stack is located in slow local memory (physically in GPU DRAM) this increases the memory traffic of the already memory limited code and hence it is not suited for the limited size of local cache of contemporary GPU architectures. The amount of data read from local memory is similar for both data structures (132 MB for BVH vs. 171 MB for kd-tree) but the L1 cache hit ratio of the read accesses is very different (90% for BVH vs. 55% for kd-tree). The amount of utilized read data is probably similar because early ray termination of the kd-tree traversal algorithm may leave some stack items unread. The different cache hit ratio is possibly caused by different threads in the same warp reading different stack indices from global memory, which causes incoherent accesses to GPU DRAM. We can only speculate that faster and larger local memory and/or cache needed by stack-based traversal algorithms could be beneficial for performance in upcoming GPU architectures for both BVHs and kd-trees.

We also performed profiling on the Powerplant scene where the kd-tree performs significantly less intersection tests and traversal steps than the BVH. The profiled statistics reveal one notable difference to the Happy Buddha scene (for which the number of traversal steps and intersections tests is very similar for both data structures). The amount of data transferred from the GPU memory to the on-chip memory (cores) is much smaller for the kd-tree due to much fewer nodes and triangles being accessed during the ray traversal algorithm (2250 MB for BVH vs. 1416 MB for kd-tree). The other profiler statistics cannot outweigh the significant difference in the amount of requested memory, explaining the lower performance of the BVH.

6.2. Hybrid data structures

In Table 5, we present the profiling statistics for SBVHs as an ratio of the BVH value. These statistics are somewhat harder to construe since the ratio of the number of splits using spatial subdivision to the number of splits using the object subdivision and the impact of spatial splits on the ray tracing performance are scene-dependent for the SBVH.

SBVHs execute 8–64% less floating point operations than BVHs because of the decreased number of performed intersection tests. Still, this is significantly more work than done by the kd-tree. The percentage of divergent branching is usually also decreased since the spatial splits produce nodes with smaller spatial extent that are less likely to be visited by rays. On the other hand, for scenes with excessive overlap of bounding boxes (Office House), where both child nodes are usually visited for BVHs, the spatial splits increase the branching divergence.

The memory related statistics ($LoadG$, $StoreL$ and $LoadL$) are almost exclusively decreased for the SBVH and usually so by a similar percentage. This indicates that the decreased number of intersection test causes less nodes to be loaded from the global memory and also stored and loaded from the traversal stack. Unlike for the kd-tree this does not come at the cost of more incoherent memory accesses or larger size of items stored to the stack. The L1 cache hit ratio was kept almost the same for SBVHs as for BVHs because the same traversal algorithm and memory layout is used for both.

7. Conclusion and Future Work

In this paper, we have compared the performance of the kd-trees and the BVHs on today's GPUs by NVIDIA with Kepler core (year 2012/2013). The number of the traversal steps and the intersection tests for ray tracing is lower for the kd-trees than for the BVHs, while the ray tracing performance is typically higher for the BVHs than for the kd-trees. In particular, the BVHs are faster on small to medium-sized scenes. This is highly important for rendering on devices with low compute power, such as mobile devices, that are capable of rendering only small scenes. On the other hand, for larger scenes with high depth complexity the kd-trees outperform the BVHs as the traversal overhead of the BVHs is significant for spatially overlapping regions. The hybrid SBVH data structure presents the best of both worlds by keeping the good hardware mapping of the BVH, while decreasing the number of intersection tests as in the kd-tree. The performance difference between the data structures can be best explained by the varying amount of data transferred from the relatively slow global memory due to the different node and triangle layouts.

Our measurements and conclusions hold for a simple memory layout model of the trees representing the BVHs and the kd-trees, i.e. each node is allocated in the memory irrespective of the memory addresses of the other nodes using a simple memory allocator. The future work could address the comparison of both data structures with memory layout optimized for decreased memory traffic of contemporary GPUs, where nodes with high spatial locality are organized into memory chunks such as in [Szé03].

Acknowledgements

We would like to thank Marko Dabrovic for the Sibenik model, DAZ3D for the Fairy Forest model, Frank Meinl at Crytek for the Crytek Sponza model, Greg Ward for the Conference model, Carlo H. Séquin for the Sodahall model, Samuli Laine and Tero Karras for the Hairball model, Guillermo Llaguno for the San Miguel model, the UNC for the Powerplant model and Stanford repository for the Dragon, Happy Buddha and Asian Dragon models.

We would also like to thank Tero Karras, Timo Aila and Samuli Laine for releasing their GPU ray tracing framework. This research was supported by the Czech Science Foundation under research programs P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, Grant No. SGS13/214/OHK3/3T/13.

References

[AKL13] Aila T., Karras T., Laine S.: On quality metrics of bounding volume hierarchies. In *Proceedings of HPG* (Anaheim, July 2013), ACM SIGGRAPH/Eurographics, pp. 101–107.

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of HPG* (New Orleans, 2009), ACM SIGGRAPH/Eurographics, pp. 145–149.

[AM05] Akenine-Möller T.: Fast 3D triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses* (2005), ACM.

[ASK14] Áfra A. T., Szirmay-Kalos L.: Stackless multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140.

[BHH13] Bittner J., Hapala M., Havran V.: Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100.

[CKL*10] Choi B., Komuravelli R., Lu V., Sung H., Bocchino R. L., Adve S. V., Hart J. C.: Parallel SAH k-D tree construction. In *Proceedings of HPG* (Saarbruecken, 2010), ACM SIGGRAPH/Eurographics, pp. 77–86.

[DPS10] Danilewski P., Popov S., Slusallek P.: *Binned SAH Kd-Tree Construction on a GPU*. Tech. Rep., Computer Graphics Group, Saarland University, June 2010.

[EG07] Ernst M., Greiner G.: Early split clipping for bounding volume hierarchies. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (Ulm, 2007), IEEE Computer Society, pp. 73–78.

[GHFB13] Gu Y., He Y., Fatahalian K., Blelloch G.: Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of HPG* (Anaheim, July 2013), ACM SIGGRAPH/Eurographics, pp. 81–88.

[GPSS07] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (Ulm, 2007), IEEE Computer Society, pp. 113–118.

[GS87] Goldsmith J., Salmon J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[Gut14] Guthe M.: Latency considerations of depth-first GPU ray tracing. In *Proceedings of Eurographics (Short Papers)* (Strasbourg, 2014), E. Galin and M. Wand M (Eds.), Eurographics Association, pp. 53–56.

[Hai87] Haines E. A.: A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*, 11 (1987), 3–5.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, November 2000.

[HB02] Havran V., Bittner J.: On improving KD-trees for ray shooting. *Journal of WSCG 10*, 1 (2002), 209–216.

[HH11] Hapala M., Havran V.: Review: Kd-tree traversal algorithms for ray tracing. *Computer Graphics Forum 30*, 1 (2011), 199–213.

[HKRS02] Hurley J., Kapustin A., Reshetov A., Soupikov A.: Fast ray tracing for modern general purpose CPU. (Nizhny Novgorod) In *Proceedings of Graphicon* (Nizhny Novgorod, 2002), p. 8.

[HMS06] Hunt W., Mark W., Stoll G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (Salt Lake City, Sept. 2006), IEEE Computer Society, pp. 81–88.

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive K-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (Seattle, 2007), ACM, pp. 167–174.

[HZDS09] Havran V., Zajac J., Drahokoupil J., Seidel H.-P.: *MPII Building Model as Data for Your Research*. Res. rep. MPI-I-2009-4-004, MPI Informatik, Dec. 2009.

[KA13] Karras T., Aila T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of HPG* (Anaheim, July 2013), ACM SIGGRAPH/Eurographics, pp. 89–99.

[Kap85] Kaplan M.: Space-tracing: A constant time ray-tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes* (July 1985), Addison Wesley, pp. 149–158.

[KIS*12] Kopta D., Ize T., Spjut J., Brunvand E., Davis A., Kensler A.: Fast, effective BVH updates for animated scenes. In *Proceedings of the I3D Conference* (Costa Mesa, 2012), ACM, pp. 197–204.

[Lee10] Lee V. E. A., Lee V. W., Kim C., Chhugani J., Deisher M., Kim D., Nguyen A. D., Satish N., Smelyanskiy M., Chennupaty S., Hammarlund P., Singhal R., Dubey P.: Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, 2010), ACM, pp. 451–460.

[ML03] Masso J. P. M., Lopez P. G.: Automatic hybrid hierarchy creation: A cost-model based approach. *Computer Graphics Forum 22*, 1 (2003), 5–13.

[NVI12]NVIDIA: NVIDIA Nsight ver 2.2. NVIDIA developer zone, 2012.

[PGDS09] Popov S., Georgiev I., Dimov R., Slusallek P.: Object partitioning considered harmful: Space subdivision for BVHs. In *Proceedings of HPG* (New Orleans, 2009), ACM SIGGRAPH/Eurographics, pp. 15–22.

[RPC12] Roccia J.-P., Paulin M., Coustet C.: Hybrid CPU/GPU KD-tree construction for versatile ray tracing. In *Eurographics (Short Papers)* (2012), C. Andújar and E. Puppo (Eds.), Eurographics Association, pp. 13–16.

[SFD09] Stich M., Friedrich H., Dietrich A.: Spatial splits in bounding volume hierarchies. In *Proceedings of HPG* (New Orleans, 2009), ACM SIGGRAPH/Eurographics, pp. 7–13.

[Szé03] Szécsi L.: An effective implementation of the k-D tree, *Graphics Programming Methods*. Charles River Media, Inc., Rockland, MA (2003), pp. 315–326.

[VBHH13] Vinkler M., Bittner J., Havran V., Hapala M.: Massively parallel hierarchical scene processing with applications in rendering. *Computer Graphics Forum 32*, 8 (2013), 13–25.

[VH14] Vinkler M., Havran V.: Register efficient memory allocator for GPUs. In *Proceedings of High-Performance Graphics* (Lyon, 2014), I. Wald and J. Ragan-Kelley (Eds.), Eurographics Association, pp. 19–27.

[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (Ulm, Sept. 2007), IEEE Computer Society, pp. 33–40.

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing* (Salt Lake City, Sep. 2006), IEEE Computer Society, pp. 61–69.

[WHG84] Weghorst H., Hooper G., Greenberg D. P.: Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*, 1 (1984), 52–69.

[WK06] Wächter C., Keller A.: Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (2006), Eurographics Association, pp. 139–149.

[WMG*09] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722.

[Woo04] Woop S.: *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Master's thesis, Saarland University, Saarbruecken, Germany, March 2004.

[WZL11] Wu Z., Zhao F., Liu X.: SAH KD-tree construction on GPU. In *Proceedings of HPG* (Vancouver, 2011), ACM SIGGRAPH/Eurographics, pp. 71–78.

[ZH10] Zlatuška M., Havran V.: Ray tracing on a GPU with CUDA—Comparative study of three algorithms. In *Proceedings of WSCG' 2010, Communication Papers* (Pilsen, Feb. 2010), pp. 69–76.