Chapter 13
On the Efficient Implementation of a Real-time Kd-tree Construction Algorithm[1]

Byungjoon Chang Woong Seo Insung Ihm

Department of Computer Science and Engineering, Sogang University, Seoul, Korea
e-mail: {jerrun,wng0620,ihm}@ sognag.ac.kr

Abstract: The kd-tree is one of the most commonly used spatial data structures for a variety of graphics applications because of its reliably high acceleration performance. Several years ago, Zhou et al. devised an effective kd-tree construction algorithm that runs entirely on a GPU. In this chapter, we present improved GPU programming techniques for implementing the algorithm more efficiently on current GPUs. One of the major ideas is to reduce the number of necessary kernel functions by replacing the essential, segmented-scan, and reduction computations by simpler per-block atomic operations, thereby alleviating the overheads from multiple synchronous kernel calls. Combined with the efficient implementation of intrablock scan and reduction, using recently introduced intrinsic functions, these changes achieve remarkable performance enhancement to the kd-tree construction process. Through an example of real-time ray tracing for dynamic scenes of non-trivial complexity, we demonstrate that the proposed GPU techniques can be exploited effectively for various real-time applications.

13.1 Background and our contribution

For many important applications in computer graphics, such as ray tracing and those relying on particle-based computations, adopting a proper acceleration structure will affect their run-time performance greatly. Among the variety of spatial data structures, the kd-tree is frequently used because of its reliably high acceleration performance. Compared to other techniques such as grids and bounding-volume hierarchies, its relatively higher construction cost has been regarded as a drawback, despite efforts to develop an optimized algorithm (e.g., [9]), which has often restricted the use of the kd-tree for real-time applications.

[1] Submitted to *2013 Symposium on GPU Computing & Applications*.

Recently, much effort has gone into accelerating kd-tree construction, particularly by developing effective parallel algorithms on modern CPUs and GPUs. Shevtsov et al. [7] and Zhou et al. [11] presented parallel construction algorithms for the CPU and GPU, respectively, in which, instead of applying a precise surface area heuristic (SAH) metric, median-splitting schemes were used to build the upper levels of the trees to enable effective parallelization on the respective processors. To alleviate memory usage issues, Hou et al. improved Zhou et al.'s method by modifying the kd-tree construction order [3]. In another approach, Choi et al. [1] and Wu et al. [10] attempted to build better kd-trees for the CPU and GPU, respectively, by applying the accurate SAH metric to the entire tree structure. As pointed out in [1], the approximate approaches taken in [7, 11] may often lead to kd-trees of somewhat degraded quality, which would influence the kd-tree performance adversely. However, for interactive applications such as the real-time ray tracing of dynamic scenes, where the kd-tree must be rebuilt for every frame after ray tracing the scene, it is important to adopt an effective kd-tree construction scheme that achieves a balance between tree-construction efficiency and run-time acceleration performance.

In this chapter, we present enhanced CUDA programming techniques for implementing the GPU method of Zhou et al. [11]. While their detailed algorithm, proposed several years ago, is still effective, current GPU designs enable it to be implemented more efficiently. In developing this CUDA implementation, we aim to enhance the GPU performance, particularly by minimizing the overheads caused by multiple synchronous kernel calls. For this, the essential, segmented-scan, and reduction computations are replaced by simpler per-block atomic operations. Coupled with an efficient implementation of intrablock scan and reduction, based on recently introduced intrinsic functions of the CUDA API, our methods achieve significant performance improvements in the kd-tree construction process. Via experiments on ray tracing for dynamic scenes of nontrivial complexity, we demonstrate that the proposed GPU techniques can be applied effectively to various real-time applications.

13.2 Optimizations for the large-node stage

In Zhou et al.'s method, the upper levels of the kd-tree were constructed using a node-splitting scheme that comprised spatial median splitting and empty-space maximizing. In particular, based on the observation that the assumptions made in the SAH may often be inaccurate for large nodes, this stage of computation, called the *large-node stage*, simply selects the spatial median of the longest axis of the axis-aligned bounding box (AABB) of a node as its split position. For efficient parallel implementation on a GPU, all triangles in each large node are grouped in-

to *chunks* of fixed size (i.e., 256), parallelizing the computation over the triangles in the chunks. (Note that the triangles and chunks are mapped to the threads and blocks, respectively, in the CUDA implementation.)

13.2.1 Triangle sorting with respect to splitting planes

The large-node stage iterates the node-splitting process until no large node is left. In Algorithm 2 [11], the most time-consuming parts of each iteration are the fourth and fifth steps, corresponding to lines 24-34 and 35-40, respectively, where the triangles for each large node are first sorted with respect to the splitting plane, and the triangle numbers of the resulting two child nodes are then counted. In this subsection, we present two different approaches to implementing these two steps on a GPU. We then analyze their performance in the section on experimental results.

13.2.1.1 Implementation using standard data-parallel primitives

As was done in [11], the first implementation relies on standard data-parallel primitives such as (segmented) scan and reduction, but uses a slightly different algorithm, which is computationally as efficient as the original one. The topmost part of Figure 13.1 shows a situation where triangles in each large node are sorted into two child nodes. Here, we allocate two lists statically, *active list* and *next list*, to the global memory of the GPU to buffer the triangle indices. (Note that the triangle indices are grouped into chunks of size 256, as shown in the dashed boxes, which are then packed into the triangle index lists.)

For each triangle in a large node, mapped to a CUDA thread, the key issue is how to efficiently calculate its address(es) in parallel in the new triangle index list *next list*, whose production is complicated because of the simultaneous subdivisions of the large nodes in the current list *active list*. For this, a kernel is first executed over every thread block corresponding to a chunk of triangles, classifying each triangle against the respective splitting plane, and generating two bit-flag sequences of size 256 per chunk *triangle bit flags*. Then, for each of these, an exclusive scan is performed using the shared memory of the GPU, resulting in the local *triangle offset* sequences. In addition, the kernel counts the number of triangles in each bit-flag sequence by simple addition, and places this number in an array in the global memory. (Note that, for the example in Figure 13.1, the two triangle counts of 201 and 75 are written to the array marked **[A]** as a result of execution over the first chunk of *node 0*.)

The next kernel then starts performing an inclusive *segmented* scan over this array, storing the scanned result in another array, marked **[B]** in the example figure, where each child node now comprises a segment in the sequence. After this scan, a per-element subtraction is carried out in parallel between these two arrays to build another *chunk offset* sequence that stores the displacement of the first triangle in each chunk within a new child node. In the subsequent third kernel, an exclusive *segmented* scan is carried out over the sequence of numbers formed by the last element of each child node in the scanned array **[B]**, whose resulting *node offsets* indicate the offsets of the first triangles of the new nodes within the new triangle index list. Finally, a fourth kernel is executed over the thread blocks of triangles in the *triangle bit flags* array, where, for a triangle whose bit flag is on, its triangle index is stored in the appropriate place in the new triangle index list, whose address can be calculated using the *node offsets*, *chunk offsets*, and *triangle offsets*.

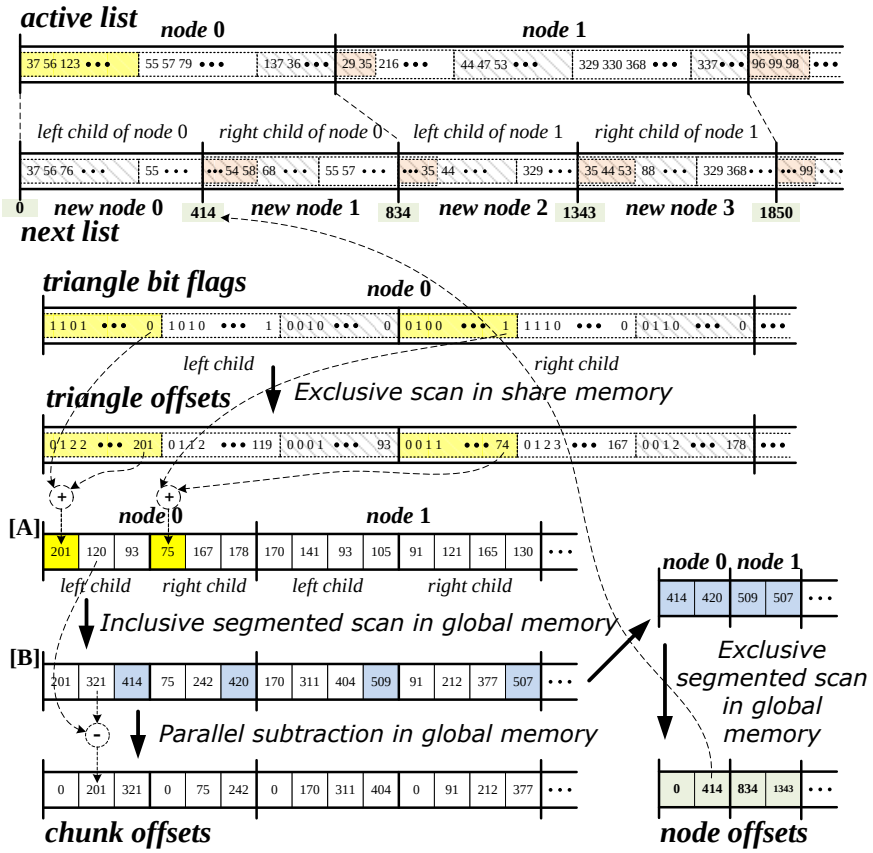


Fig. 13.1. Parallel triangle sorting over splitting planes: using standard segmented-scan primitives.

13.2.1.2 Implementation using atomic operations

The triangle-sorting technique described in the previous subsection requires a segmented scan to be carried out twice on the data sequences stored in the global memory, and can easily be implemented using the data-parallel primitive functions provided by the CUDPP library [2], for example. Although very effective, such an approach forces the run-time execution to be split into a sequence of synchronous kernel calls, whose overheads will impact the run-time performance adversely.

To address this, observe that a side effect of using a standard segmented-scan method is that the relative order of triangle indices within a large node made of multiple chunks is retained in the respective child nodes. Such a property is important when the order of elements is essential, as in a radix sort algorithm, for example. However, retaining the strict order is unnecessary in the kd-tree construction algorithm because the order of triangles within a kd-tree's leaf node is not critical in the later ray-tracing stage. This observation allows us to implement the triangle-sorting computation by using a single faster-running kernel and replacing the segmented-scan operations with simpler per-chunk atomic operations that are supported by the CUDA API.

In the new implementation, the memory configuration for the triangle index lists is slightly different, as shown in Figure 13.2. For the ith large node with n_i triangles in the current *active list*, $2n_i$ elements, n_i per child node, are consecutively allocated to the *next list*. In addition, an array of integer-valued *chunk offset* counts, all initially set to zero, is allocated in the global memory, each of whose elements corresponds to a child node, i.e., a new node in the *next list*. As before, these *atomic* variables are intended to hold the displacements of the first triangles in the chunks within a new child node, although the order between chunks may no longer be preserved because of the use of the atomic operation.

For each chunk of triangle indices in the current list, the new kernel repeats the same computation until the triangle numbers are calculated in the array **[A]**. A representative thread then carries out two atomic additions, respectively fetching the local offsets, one for each child node, from the corresponding atomic variables and simultaneously adding the triangle counts to them, through which we will know where to start storing the sorted triangle indices in the child nodes. Then, once per child node, each thread checks the corresponding bit flag in the *triangle bit flag array*, and, if set to *on*, puts its triangle index in the proper place in the next triangle index list, whose location can easily be deduced from the fetched offset and the offset in the *triangle offsets* array.

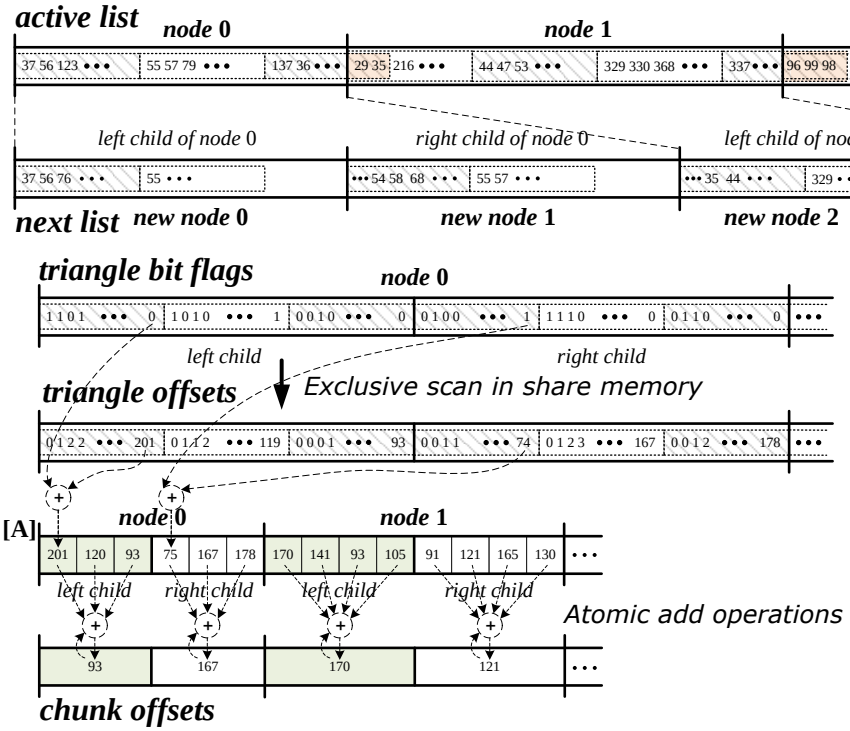


Fig. 13.2. Parallel triangle sorting over splitting planes: using atomic add operations.

.

In this implementation, the two segmented scans over the arrays in the global memory have been replaced by two atomic-add operations per thread block. While the computation time is already reduced markedly by this change, two per-block scans, one for each child, must still be carried out per chunk to compute the triangle offsets. While such scans can be performed effectively in the shared memory by using a standard scan method [6], recent GPUs offer useful intrinsic operations, such as __ballot() for warp voting, __popc() for bit counting, and __shfl() for warp shuffling, that can enable an efficient implementation of the per-block scan [5]. Therefore, to achieve a further performance enhancement, our implementation uses the __ballot() and __popc() functions for an intra-warp scan [8], and the __shfl_up() function for an inter-warp scan. (Details of our CUDA implementation of the kernel function are described in the Appendix.)

13.2.2 AABB computations for active large nodes

Another time-consuming part of the large-node stage is the second step (lines 9 to 14 of Algorithm 2), in which the AABB of all triangles in each node is calculated. The optimization techniques described in the previous subsection can also be applied to this AABB computation. The standard reduction in the shared memory for computing per-chunk bounding boxes can be implemented more efficiently on the GPU by a simple modification of the scan implementation using the intrinsic shuffle function __shfl_up(). Then, via three pairs of atomic *min* and *max* operations, the result of each chunk reduction is written in parallel to the location in the global memory that corresponds to the large node to which the chunk belongs. Although such atomic operations are still regarded as expensive on current GPUs, we observe that our single-kernel implementation based on atomic operations runs significantly faster on the GPU than the original implementation, which needed to perform segmented reductions six times.

13.3 Optimizations for the small-node stage

After all large nodes are split into nodes whose triangle numbers do not exceed 64, the *small-node stage* starts. Because sufficient nodes are available, the computation in this stage is parallelized over nodes instead of triangles, evaluating the precise SAH metric to find the best splitting plane for each small node. The key to the efficient implementation of this stage is exploiting a preprocessed data structure that facilitates the iterative node-splitting process. For each initial small node, called the *small root node*, up to 384 (= 64 (triangles) * 3 (x-, y-, z-axes) * 2 (min/max)) splitting-plane candidates are first collected from triangles in the node. Then, for each candidate, two 8-byte bit masks are generated to represent the triangle sets contained in both sides. To represent this information, 20 bytes of memory per node is necessary, including the 4 bytes used to store the location of the splitting plane, implying that up to 7,680 (=20 * 384) bytes of memory may be necessary for each small root node. It is important to choose an appropriate memory layout for the representation because the nontrivial amount of data will be accessed in parallel during the small-node stage. Although several different configurations are possible, we observed that the combination of a 4-byte access from the global memory for the splitting plane location and another 16-byte access from the texture memory for the triangle sets incurred the lowest memory latency on the GPU tested. (Our analysis of the generated PTX code showed that 16 bytes of data were fetched from texture memory even for a 4-byte access command.)

With this representation, the SAH cost evaluation and triangle sorting in the subsequent node-splitting step can be performed efficiently using simple bitwise

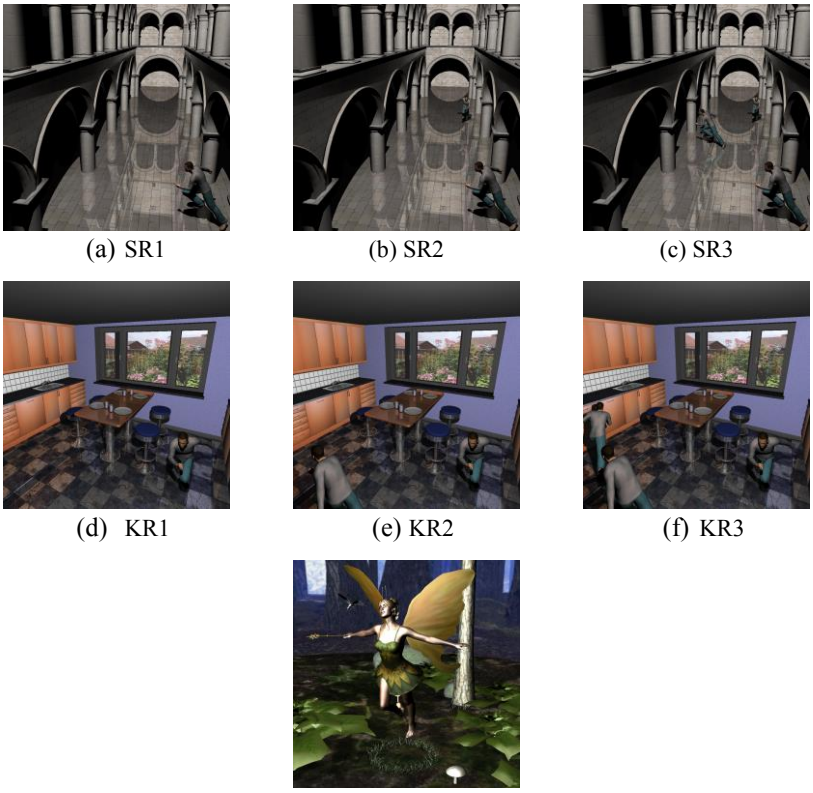
operations. In this process, a parallel bit-counting operation is carried out very frequently to obtain the numbers of triangles in the child nodes. Whereas the method presented in [4] was used in the original description of Zhou et al.'s algorithm, we find that the __popc() intrinsic function accelerates the counting process significantly, as will be shown in the next section. Furthermore, we can also accelerate the intrablock scan, using the same intrinsic functions as for the triangle-sorting computation, which improves the performance slightly.

13.4 Experimental results

To measure the performance improvement achieved by the optimization techniques presented here, we first implemented the kd-tree construction algorithm of Zhou et al. on an NVIDIA GeForce GTX 680 GPU, effectively as described in the original paper. In doing this, we used the scan and reduction techniques described in [6] for both *intra-chunk* and *segmented* scan and reduction. Here, the CUDPP primitive functions [2] were utilized for the segmented data. Furthermore, the parallel bit counting operation needed in the small-node stage was implemented as proposed in [4]. Starting with this original implementation, we applied the optimization techniques described above one at a time, in the order given in Table 13.1, and measuring their impact on the timing performance. Note that the order of triangle indices in the leaf nodes of the produced kd-trees may be different because of the simultaneous atomic operations performed in our method. To check for any effects on rendering performance, we also measured the time to render a 1024 * 1024 image by full ray tracing with shading, textures, reflection, and shadows. To experiment with dynamic scenes of nontrivial complexity, we synthesized some test scenes from commonly used scenes, made available by courtesy of the Utah 3D Animation Repository, Joachim Helenklaken, and Marko Dabrovic: “Sponza with i Runners” (SRi) and “Kitchen with i Runners” (KRi), for i = 1, 2, 3. These scenes comprise 66,454 + i * 78,029 triangles and 101,015 + i * 78,029 triangles, respectively. (See Figure 13.3.)

For 7 representative scenes, Table 13.2(a) gives the stage-by-stage reduction in the kd-tree construction time, achieved as a result of the application of the series of optimization techniques described above. It is clear that the replacement of segmented scan and reduction by per-block atomic operations produced significant improvements despite atomic operations still being regarded as costly in the current CUDA architecture. (See the changes from “Original” to “[A]” and “[A] - [B]” to “[A] - [C]”.) A major reason is that the operations explained in Sections 2.1 and 2.2, respectively, were able to be executed more efficiently on the GPU using fewer numbers of kernels, as clearly indicated in the “Kernel calls” row, which markedly reduced the overheads from multiple synchronous kernel calls. (Note that a single kernel was sufficient for the triangle-sorting process, while four

plus those necessary for the two segmented-scan calls were needed, which was repeated per each iteration.) Also, by exploiting the intrinsic functions offered by the more recent CUDA compute capability, we could reduce the computation cost for the intrablock scan and reduction further.



(a) SR1 (b) SR2 (c) SR3

(d) KR1 (e) KR2 (f) KR3

(g) FF

Fig. 13.3. Test scenes. The numbers of triangles in these scenes are 144,483(SR1), 222,512(SR2), 300,541(SR3), 179,044(KR1), 257,073(KR2), 335,102(KR3), and 174,117(FF).

Table 13.1. Applied optimization techniques. In (a), LNS1 and LNS2 denote the computation for triangle sorting (Sec. 2.1) and AABB computation (Sec. 2.2), respectively, in the large-node stage, while SNS denotes the small-node stage (Note that † refers to the intra-warp scan of binary numbers and ‡ refers to the inter-warp scan).

	Stage	Operations	Our implementation
[A]	LNS1	Two segmented scans	two atomic operations per block
[B]	LNS1	Intra-block scans	__ballot( )/__popc( ) † __shfl_up( ) ‡
[C]	LNS2	Six seg. reductions	six atomic operations per block
[D]	LNS2	Intra-block reductions	__shfl_up( )
[E]	SNS	Intra-block scans	__popc( ) same as [B]

As can be verified from the ray-tracing time "R" in Table 13.2(b), the modifications to the original kd-tree construction algorithm did not incur any noticeable degradation in the quality of generated trees except a few cases, despite the different orders of triangle indices stored in the leaf nodes. As a result, we were able to accelerate the process of interactive ray tracing of nontrivial dynamic scenes on the GPU effectively. (This is shown by the "T" values in Table 13.2(b).)

13.5 Concluding remarks

In this chapter, we have presented efficient GPU programming techniques for implementing the well-known kd-tree construction algorithm [11], and demonstrated its effectiveness through several examples. With current GPUs, executing a CUDA kernel is still a relatively expensive operation, and thus it is important to make an effort to minimize the number of kernel calls made. As shown in the result section, our method was shown to be very successive in building kd-trees using much fewer numbers of kernels, which resulted in a markedly more efficient GPU implementation. We believe that the ideas presented are also relevant to the development of applications that use other hierarchical spatial data structures.

Acknowledgments: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (**MOE**) (No. 2012R1A1A2008958).

Table 13.2. Performance of the kd-tree construction. In (a), we provide the total time spent on the kd-tree construction, averaged for given animation sequences, where the number in parentheses represents the timing obtained by summing each kernel's execution time. Also, the averaged numbers of CUDA kernel calls made by the original and our implementations are compared, in which the extra calls within the CUDPP functions for the original implementation were not counted here. In (b), the average ray-tracing time (R) and the average total time (T), which includes both construction and ray tracing, are given. FF denotes the "Fairy Forest" scene comprising 174,117 triangles.

(a) Construction time (in milliseconds) and numbers of kernel calls

	SR1	SR2	SR3	KR1	KR2	KR3	FF
Original	120.9	146.2	181.8	130.9	149.7	185.5	107.4
	(46.3)	(59.1)	(77.7)	(47.1)	(60.0)	(77.8)	(42.1)
[A]	88.6	93.0	109.7	91.8	109.1	118.9	76.2
	(34.8)	(49.9)	(64.2)	(35.1)	(50.0)	(64.9)	(36.8)
[A] - [B]	76.3	89.2	107.4	85.7	94.7	104.4	67.1
	(34.1)	(49.0)	(62.8)	(34.3)	(48.6)	(62.6)	(36.2)
[A] - [C]	52.3	67.5	79.8	62.3	67.6	90.6	58.5
	(29.9)	(44.7)	(59.0)	(31.1)	(44.9)	(59.3)	(34.0)
[A] - [D]	51.6	66.4	78.1	60.1	67.1	88.6	54.5
	(29.2)	(43.3)	(56.9)	(29.8)	(43.6)	(57.4)	(32.8)
Ours(All)	48.5	64.3	74.1	48.8	64.0	72.5	48.1
	(26.7)	(39.1)	(52.4)	(27.4)	(40.6)	(52.8)	(30.4)
Kernel Calls	1,031	1,041	1,069	977	995	1,034	681
	/221	/229	/232	/214	/219	/223	/153

(b) Rendering time (in milliseconds)

		SR1	SR2	SR3	KR1	KR2	KR3	FF
Orig.	R	92.5	83.9	95.4	90.0	83.5	88.7	92.1
	T	213.4	230.1	277.2	220.9	233.2	274.2	199.5
Ours	R	90.1	92.7	93.5	88.2	94.3	100.5	103.3
	T	138.6	157.0	167.6	137.0	158.3	173.0	157.8

References

[1] Choi B, Komuravelli R, Lu V, Sung H, Bocchino R, Adve S, Hart J (2010) Parallel SAH k-D tree construction. In Proc of High Perf Graph (HPG '10), 77–86

[2] CUDPP Google Group (2011) CUDA data parallel primitives library release 2.0. http://code.google.com/p/cudpp/. Accessed 1 June 2013

[3] Hou Q, Sun X, Zhou K, Lauterbach C, Manocha D (2011) Memory-scalable GPU spatial hierarchy construction. IEEE Trans on Vis & Comp Graph 17:466–474

[4] Manku G (2002) Fast bit counting routines. http://cpptruths.googlecode.com/svn/trunk/c/bitcount.c. Accessed 1 June 2013

[5] NVIDIA (2012) CUDA C Programming Guide: Design Guide (PG-02829-001 v5.0)

[6] Sengupta S, Harris M, Garland M, Owens J (2011) Efficient parallel scan algorithms for many-core GPUs. In Scientific Computing with Multicore and Accelerators, Taylor & Francis, 413–442

[7] Shevtsov M, Soupikov A (2007) Highly parallel fast Kd-tree construction for interactive ray tracing of dynamic scenes. Comp Graph Forum (Proc of Eurographics) 26:395–404

[8] Skjellum A, Whittaker D, Bangalore P (2010) Ballot counting for optimal binary prefix sum. Presented in the GPU Tech Conf 2010

[9] Wald I, Havran V (2006) On building fast kd-trees for ray tracing, and on doing that in O(Nlog N). In Proc of the IEEE Symp on Interactive Ray Tracing, 61–69

[10] Wu Z, Zhao F, Liu X (2011) SAH KD-tree construction on GPU. In Proc of High Perf Graph (HPG '11), 71–78

[11] Zhou K, Hou Q, Wang R, Guo B (2008) Real-time KD-tree construction on graphics hardware. ACM Trans on Graph 27:1–11

Appendix: A single kernel implementation for the triangle sorting process (Section 13.2.1.2)

```
/* This kernel corresponds to the fourth and fifth steps of the large node stage described in [11]. */

__global__ void MedianSplitChunk(float *TriAABB, int *ChunkNodeIDs,
                                 int *NodeTriOffsets, int *NodeTriNums,
                                 int *ChunkStartIndices, int *ActiveNodeList,
                                 char *NodeSplitAxes, float *NodeSplitPoss,
                                 int *ChunkOffsets, int *NextNodeList) {

    __shared__ volatile int LChildTriOffsets2[9], RChildTriOffsets2[9];
    __shared__ int LChildNodesID, RChildNodesID, LOffset, ROffset;

    int LaneID = threadIdx.x & 0x0000001f;
    int NodeID = ChunkNodeIDs[CurBlockIndex];
    int CurBlockIndex = blockIdx.x
    int TriNum = NodeTriNums[NodeID], TriOffset = NodeTriOffsets[NodeID];
    int LChildTriOffsets, RChildTriOffsets;

    if (threadIdx.x < 9)
        LChildTriOffsets2[threadIdx.x] = RChildTriOffsets2[threadIdx.x] = 0;

    int TriIndex, StartPos = ChunkStartIndices[CurBlockIndex];
    int CurPos = StartPos + threadIdx.x;

    /* Classify the current triangle w.r.t. splitting plane. */
    unsigned int LChildTriBitFlag = 0, RChildTriBitFlag = 0;

    if (CurPos<TriNum) {
        /* The last chunk may have fewer than 256 triangles. */
        int SplitAxis = NodeSplitAxes[NodeID];
        float SplitPos = NodeSplitPoss[NodeID];

        TriIndex = ActiveNodeList[TriOffset+ CurID];
        float MinPos = TriAABB[TriIndex + SplitAxis * TRI OFFSET];
        float MaxPos = TriAABB[TriIndex + (SplitAxis + 3) * TRI OFFSET];
        LChildTriBitFlag = (MinPos < SplitPos);
        RChildTriBitFlag = (MinPos >= SplitPos);
        if (LChildTriBitFlag)
                RChildTriBitFlag = (SplitPos < MaxPos);
    }

    /* Perform intra-warp scan. */
    unsigned int LeftMask = ballot(LChildTriBitFlag), LaneMaskLT = 0;
```

```
unsigned int RightMask = ballot(RChildTriBitFlag), LaneMaskLE = 0;

asm("mov.u32 %0, %%lanemask lt;" : "=r"(LaneMaskLT));
asm("mov.u32 %0, %%lanemask le;" : "=r"(LaneMaskLE));

LChildTriOffsets = popc(LeftMask & LaneMaskLT);
RChildTriOffsets = popc(RightMask & LaneMaskLT);

if (LaneID == 31) {
    LChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(LeftMask & LaneMaskLE);
    RChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(RightMask & LaneMaskLE);
}
syncthreads();

/* Perform inter-warp scan. */
float Scan8[2];
if (threadIdx.x < 8) {
    Scan8[0] = LChildTriOffsets2[threadIdx.x + 1];
    Scan8[1] = RChildTriOffsets2[threadIdx.x + 1];

    for (int i = 1; i <= 4; i *= 2) {
        float n0 = shfl up(Scan8[0], i, 8);
        float n1 = shfl up(Scan8[1], i, 8);

        if (LaneID >= i) {
            Scan8[0] += n0;
            Scan8[1] += n1;
        }
    }
}

if (threadIdx.x < 8) {
    LChildTriOffsets2[threadIdx.x + 1] = Scan8[0];
    RChildTriOffsets2[threadIdx.x + 1] = Scan8[1];
}

/* Fetch start positions for the current chunk. */
if (threadIdx.x == 0) {
    LChildNodesID = 2*NodeID; RChildNodesID = 2*NodeID + 1;

    LOffset = atomicAdd(&ChunkOffsets[LChildNodesID], LChildTriOffsets2[8]);
    ROffset = atomicAdd(&ChunkOffsets[RChildNodesID], RChildTriOffsets2[8]);
}
syncthreads();

LChildTriOffsets += LChildTriOffsets2[(threadIdx.x >> 5)];
```

```
    RChildTriOffsets += RChildTriOffsets2[(threadIdx.x >> 5)];

    if (LChildTriBitFlag != 0)
        NextNodeList[LOffset + LChildTriOffsets] = TriIndex;

    if (RChildTriBitFlag != 0)
        NextNodeList[ROffset + RChildTriOffsets] = TriIndex;
}
```