Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing

Jean-Patrick Roccia[1,2], Mathias Paulin[1], Christophe Coustet[2]

[1]IRIT - Université de Toulouse, France
[2]HPC-SA, Toulouse, France

Abstract

We propose an hybrid CPU-GPU ray-tracing implementation based on an optimal Kd-Tree as acceleration structure. The construction and traversal of this KD-tree takes benefit from both the CPU and the GPU to achieve high-performance ray-tracing on mainstream hardware. Our approach, flexible enough to use only a single computing unit (CPU or GPU), is able to efficiently distribute workload between CPUs and GPUs for fast KD-tree construction and traversal.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Modern ray-tracing applications require tracing a huge number of rays in the same scene. However, the coherence of rays is rarely sufficient to allow their easy and efficient packetization. Monte Carlo based methods, designed to numerically solve integral equations using a high number of independent samples, typically fall in this category. In the case of the rendering equation, samples are rays or paths made of rays. Such methods are widely used in radiative simulation or high quality rendering and their accuracy is directly dependent on the number of rays traced. Nevertheless, all the applications do not require a huge number of rays to provide an accurate result. For instance, for interactive exploration or product design, the user just needs to evaluate an equation involving a few rays to get a virtual measure at one point or to select one object in the scene.

In order to fulfill constraints of both types of applications, we propose two contributions for building an optimal KD-Tree: a technical one, *event encoding*, that reduces the building cost of a KD-Tree and a system one, *hybrid CPU/GPU architecture*, that efficiently balances the computations between CPU and GPU.

As demonstrated in section 2, current parallel algorithms for building KD-Trees do not lead to optimal trees or are memory limited. We propose in section 3.1 an encoding of the split-events that limits the memory consumption of the tree and in section 3.2 an efficient hybrid CPU/GPU algorithm that minimizes data transfers as well as computation time. As our KD-Tree construction is hybrid both the GPU and the CPU can use the tree for efficient traversal (section 4). We then demonstrate the versatility of our system when it is implemented in CUDA (section 5).

2. Previous works

While the number of computation cores available on CPU or GPU increases, many parallel approaches for building KD-Trees are proposed. Nevertheless, increasing parallelism and improving load-balancing often lead to sub-optimal trees.

The construction of a KD-Tree consists in splitting a node at a location that balances the tree according to a cost function. Surface Area Heuristic (SAH) [MB90] is known to be very efficient for ray tracing. Due to the lack of parallelism on the top-level nodes and in order to quickly generate enough nodes for subsequent parallel computations, split heuristic is often simplified in the first steps of the construction. This improves the construction time, but it has two drawbacks: first, the KD-Tree is not optimal on the top nodes and, second, increasing the number of computing units forces the use of a simplified split heuristic for an increasing number of nodes.

There are two common simplifications for split finding methods. The first one consists in using the initial bounding boxes of triangles to evaluate the SAH cost of all split

© The Eurographics Association 2012.

DOI: 10.2312/conf/EG2012/short/013-016

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

candidates. This implies that the selected split plane is not necessarily the best choice. The second one consists in only testing a constant number of regularly distributed split planes instead of using real limits of triangles. [HB02] estimates that the use of the real limits of triangles provides a 9-35% gain in traversal time compared to initial triangles boxes and demonstrates that these trees are optimal.

[WH06] proposes an improved construction algorithm for an optimal KD-Tree that [CKL*10] partially parallelizes on a four 8-cores Xeon computer (not a particularly mainstream setting). While increasing construction time, both methods improve traversal time. Recently, [WZL11] introduced a parallel algorithm for optimal KD-Tree construction based on an optimized partition-and-sort algorithm and on the parallel evaluation of the SAH. Even though highly efficient, the memory consumption limits the scalability of this approach.

The standard traversal algorithm introduced in [HKBv98] is still the more effective on the CPU thanks to early-exit optimization and segmented ray validity handling. The idea is to always traverse the first intersected child to ensure that the best valid intersection found in a visited node is necessarily the final intersection. However, a dynamic stack is required in order to come back on the last visited node when no valid intersection is found, which is not GPU friendly. For this reason, [FS05] introduces a stackless traversal approach for KD-Tree on GPU that modifies the validity range of the ray to restart from the root node when no intersection is found. [PGSS07] proposes another stackless algorithm that interconnects adjacent nodes by so-called ropes. This increases performance by accessing directly the next node to traverse instead of going back in the tree levels when no intersection is found in the current leaf. The main disadvantage of this method is the memory cost of the ropes that increases by a factor of two the memory cost of the tree.

3. Hybrid KD-Tree Construction

The aim of our building method is to leverage both the CPU and the GPU, to obtain an optimal KD-Tree, without approximation during the split plane selection. Our first contribution targets the representation of triangle limits, which are commonly named events. The second contribution targets a robust and efficient load balancing model between CPU and GPU.

3.1. Event representation

Each node contains the list of all triangle limits in its space. A triangle limit is a location value, a triangle index and an event type (start or end of triangle). This list must be kept ordered during the whole construction of the tree to allow numerous optimizations for the SAH cost evaluation. It must also be modified at each split to remove the triangles which do not lie in child nodes and to update events of shared triangles (i.e. triangles intersected by the split plane).

In [WH06] a float is used for the event location along with a bool for the event type. All the weightiness in the processing of nodes comes from this very simple choice. Ordering predicate is complex as it must ensure that start events are placed before end events in case of equal position and requires two memory accesses. Using this encoding, 6 bools per triangle are required in the node structure.

Initial value	0.0f 0x00000000	0.1f 0x3dcccccd	-0.1f 0xbdcccccd
Start event	0.0f 0x00000000	0.099999994f 0x3dcccccc	-0.1f 0xbdcccccd
End event	1.401e-45f 0x00000001	0.1f 0x3dcccccd	-0.099999994f 0xbdcccccc

Table 1: *Examples of event type encoding in the event itself.*

Device	Without event-type merging	With event-type merging	Acceleration factor
CPU	3.588s	0.796s	x4.5
GPU	0.390s	0.016s	x24.4

Table 2: *Events representation influence on sorting time. CPU Intel Core i7 920, GPU NVidia GeForce GTX 560 sorting of 12M events using [HB10].*

By finding a way to merge events with their types when manipulating the event/type couple, we can compact this encoding. This compaction must preserve ordering of events and must limit the KD-Tree sub-optimality to a bare minimum. Eventually, event types only require one bit. This is why we decided to store the event type into the least significant bit of the float value of the event location. So, an event is only a structure containing a modified float and its associated triangle index. The least significant bit of a start event is set to the sign bit of the float, when an end event receives the boolean complementation of its sign bit (see Table 1).

Depending on the float sign and event type, this can shift the float to the previous/next IEEE representable float. This leads to a minimal KD-tree perturbation, maintains event ordering, and ensures that a start event always comes before its corresponding end event, even for aligned triangles.

This encoding allows us to directly use high performance libraries providing key/value sort functions, with best performance on floats/unsigned (see Table 2), in order to sort events and their associated triangles indexes both on the CPU and the GPU. The type of an event can be retrieved by comparing its sign bit to its least significant bit, avoiding a memory access to a bool.

3.2. CPU/GPU task repartition

The goal of our repartition is to organize the collaboration between the CPU and the GPU in order to better take advantage of most of their specificities without being forced to use both of them.

to this end, nodes of the KD-Tree are divided into two

© The Eurographics Association 2012.

categories : small nodes and large nodes. This classification depends on their number of triangles, *NT*. The aim of this classification is to process the large nodes on the GPU in order to exploit massive parallelism on their high number of events, whereas small nodes are processed on the CPU. If *NT* is greater than a user-specified threshold, the node is a large node. Otherwise it is a small node. This threshold must be set according to the maximal depth of the KD-Tree and the total number of triangles in the scene. In our tests, we experimentally set this threshold to 100000 triangles to obtain the best efficiency (See figure 1 in the accompanying material).

The GPU creates the root of the kd-tree by computing and sorting the initial events. Then, it processes the large nodes in depth-first order to quickly generate small nodes that are processed by the CPU. Nodes are sent one by one to the GPU in order to reduce the maximum memory consumption. The initial sort is efficiently performed by a key/value sort on primitive types (floats/unsigned), thanks to the event compact representation proposed in section 3.1. The CPU processes small nodes in parallel with one thread per node.

This approach is very flexible in term of computing repartition: if no suitable GPU is available, the CPU builds the entire tree by considering that all the nodes are small nodes. The GPU can also consider that all the nodes are large nodes and our algorihm will run fully on GPU.

The small nodes are processed with the four steps method of [WH06] using our merged event representation. The CPU threads are waiting for any small nodes pushed in a small node buffer. They are released when the small node buffer is empty and the large nodes thread is over. Note that if a large node cannot be computed on the GPU, we just have to push this large node into the small node buffer and the CPU processes it. There are two cases where computing power is lost: when the CPU is waiting for the first small node, and when the GPU is idle, waiting for the CPU to finished small nodes processing. This idle time can be filled by implementing a large node process for the CPU, as in [CKL*10], and a small node process for the GPU, taking many small nodes and processing them in parallel.

4. Hybrid KD-Tree traversal

We have implemented two versions of the KD-Tree traversal: one using a static stack [HKBv98] and another using the stackless KD-restart method [FS05]. Both traversal algorithms do not use any paquetization or coherency classification of rays. The static stack approach is 30% faster than the KD-restart method and all our measurements are made using this algorithm.

We use a task manager to launch all the ray-tracing thread, thereafter called tracers. Our task manager can be viewed as a simple counter of remaining rays and a pointer to input (rays)/output (hits) structures. Every tracer then retrieves from the task manager the number of rays to process.

Using the NVIDIA CUDA API [NVI] zero-copy memory access capabilities, ray tracing can be distributed very simply: the same input/output pointers and an offset can be passed to all tracers independently of their types. The GPU tracers call directly a CUDA kernel with these parameters, without explicitly transferring any data. In order to avoid CPU/GPU divergence on results, the same algorithm is used to traverse the KD-Tree on both the CPU and the GPU. Load balancing between the CPU and the GPU is difficult for every hybrid method. In our system, we use a simple heuristic that first assigns rays to the CPU tracers, to avoid launching a GPU kernel for a too little number of rays. In our tests, giving the GPU one hundred times more rays than the CPU lead to the best efficiency.

The main difficulty of our approach is to find the adequate number of rays that each CPU/GPU thread requests to the task manager. This can be solved by tracing some rays at the initialization stage in order to calibrate the tracers, or by using precomputed benchmarks on the system configuration.

5. CUDA implementation

All the GPU part of our hybrid raytracer is implemented with the NVIDIA CUDA API. The KD-tree and the rays/hits buffers are shared by the CPU and the GPU using zero-copy memory access. The large node processing is split into a root initialization step and four node processing steps. Each step is totally adapted for a GPU parallelization.

Step 1: Root initialization (see Figure 1-1) consists in two phases. First, a CUDA kernel computes the min/max limits on each axis and fills the events value/index by axis. Second, events are sorted on each axis, in parallel for each triangle.

Step 2: Find best split plane (see Figure 1-2) for a node. This step takes the buffers of event values and their associated triangles index as input. For each event, the left counter buffer is initialized with 1 for start events and 0 for end events. The right counter buffer is initialized with 0 for start events and 1 for end events. A parallel exclusive scan on the left counter buffer and a reversed parallel exclusive scan on the right counter buffer gives the number of left/right triangles for each split plane candidate. The SAH cost for each split plane candidate is computed in parallel. Finally, the minimum SAH cost value is determined using a parallel minimum finding on GPU.

Step 3: Classify triangles (see Figure 1-3) as left/right/shared with respect to the best split plane found in the previous step. This step treats all start events in parallel, and classifies their corresponding triangles as left/right triangles according to the position of the event relatively to the best event. Triangles associated to end events located after the best event and classified as left triangles are then modified and re-classified as shared triangles.

© The Eurographics Association 2012.

Model	[CKL*10]		Our builder			
	1-core CPU^A	32-cores CPU^A	1-core CPU^B	4-cores CPU^B	GPU	hybrid
Dragon	5.5	0.65	1.55	0.77	2.67	0.43
Happy	6.8	0.83	1.86	0.92	2.88	0.50
Soda	/	/	4.5	2.68	2.24	1.03

Table 3: *Construction times (in seconds) on some well-known Stanford Computer Graphics Laboratory models, maximum tree depth = 8 for all measures. CPU^A: Intel Xeon X7550*4 sockets, CPU^B: Intel Core i7 920, GPU: NVidia GeForce GTX 560.*

Step 4: Filter geometry (see Figure 1-4) and compute new events. This step takes triangle flags generated by the previous step as input, and classifies events according to their associated triangle flag. Left and right triangles and events are just kept sorted for the next step. For shared triangles, for the split axis, a kernel is launched to clamp event to the split plane. For other axes, shared triangles are intersected with the split plane to generate new events. At this point, shared event lists for the left and the right side are available.

Step 5: Finalize lists (see Figure 1-5) and merge shared events with the left/right side lists. This step consists in merging the left/right events with the sorted shared left/shared right events computed in the step 4.

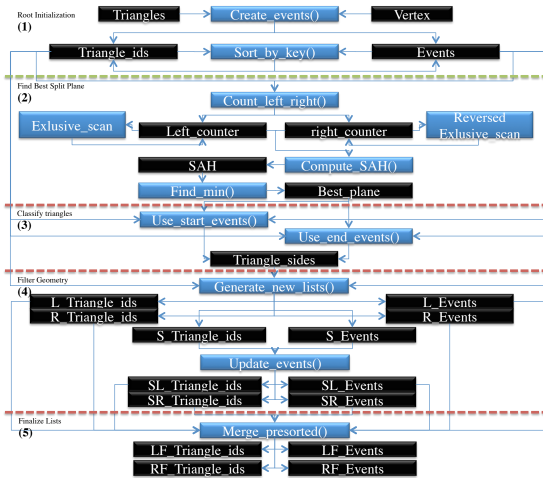

Figure 1: *GPU large node processing implementation details, step by step. Blue cells for CUDA kernels, black cells for data. L=left, R=right, S=shared, F=final.*

6. Results

Event/type merging is the central point of our KD-Tree construction algorithm. It makes the processing on mainstream hardware simpler and faster (see Table 3). The hybrid CPU/GPU organization provides a gain of 51-65% on the construction times, and outperforms the previously available HPC-class hardware performance.

Model	[AL09]	Our tracer	Acceleration
Dragon (871K)	34.5	108.5	x3.15
Happy (1M)	32.5	112.3	x3.45
Soda (2M)	32	56.0	x1.75

Table 4: *Traversal performances for four diffuse rays per pixels (in Mrays.s-1). Results obtained on NVidia GeForce GTX 560.*

We confront results with [AL09], a BVH based high performance raytracer. We use the Fermi implementation of the author, available at [KAL]. Only diffuse rays are measured to avoid coherent rays effect. Our ray tracer effectively outperforms [AL09] on this kind of rays (see table 4).

The hybrid part of the traversal allows us to launch isolated rays on the CPU, without CUDA kernel call extra cost, and obtains high performance (2 Mrays per second per core) when GPU can not be efficiently used.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics* (2009), HPG '09, ACM, pp. 145–149. 4

[CKL*10] CHOI B., KOMURAVELLI R., LU V., SUNG H., BOCCHINO R. L., ADVE S. V., HART J. C.: Parallel SAH KD-Trees construction. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG '10, Eurographics Association, pp. 77–86. 2, 3, 4

[FS05] FOLEY T., SUGERMAN J.: KD-Tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), HWWS '05, ACM, pp. 15–22. 2, 3

[HB02] HAVRAN V., BITTNER J.: On improving KD-Trees for ray shooting. *Journal of WSCG 10*, 1 (2002), 209–216. 2

[HB10] HOBEROCK J., BELL N.: Thrust: A parallel template library, 2010. Version 1.3.0. 2

[HKBv98] HAVRAN V., KOPAL T., BITTNER J., ŽÁRA J.: Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools 2* (January 1998), 15–23. 2, 3

[KAL] KARRAS T., AILA T., LAINE S.: Source code - "Understanding the efficiency of ray traversal on GPUs". http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/. 4

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Vis. Comput. 6* (May 1990), 153–166. 1

[NVI] NVIDIA: CUDA Zone - NVIDIA Developer Zone. http://developer.nvidia.com/category/zone/cuda-zone. 3

[PGSS07] POPOV S., GÃIJNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 2

[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 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 61–69. 2, 3

[WZL11] WU Z., ZHAO F., LIU X.: SAH KD-Tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), HPG '11, ACM, pp. 71–78. 2

© The Eurographics Association 2012.