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

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

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

Abstract

In this paper, 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 benefits from both the CPU and the GPU to achieve high-performance ray-tracing on mainstream hardware. Our approach, flexible enough to use only one computing units (CPU or GPU), is able to efficiently distribute workload between CPUs and GPUs for fast construction and traversal of the KD-tree.

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 high number of rays on the same scene. However, rays are rarely coherent enough to make their packetization easy and efficient.

Monte Carlo based methods, designed to numerically solve integral equations using a high number of independent samples, are typically this kind of methods. In the case of the rendering equation, samples are rays or paths made of rays. Such methods are widely used in radiative simulation and high quality rendering and their accuracy is directly dependent on the number of rays traced. Results are refined by iterative step of simulation, and each iteration of a Monte Carlo simulation generates an important number of random, thus incoherent, rays.

Nevertheless, all the applications do not rely on a huge number of rays to give a result. For instance, for interactive exploration or design, the user just needs to evaluate an equation involving 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 kind of applications, we propose **two contributions** for building an optimal KD-Tree: a technical one, **split 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-Tree do not lead to optimal trees. In order to build an optimal KD-Tree, we show in section 3.1 how to encode split-events to allow efficient hybrid CPU/GPU algorithm that minimizes data transfers as well as computation time (section 3.2). As our KD-Tree is built in a hybrid way, it can be efficiently used for traversal on both processors. We show in section 4 how versatile our system could be when using a CUDA implementation (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 building 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. However, due to the lack of parallelism on the top-level nodes and in order to quickly generate enough nodes for subsequent computations, split heuristic is often simplified in the first steps of the construction. This method clearly improves the construction time, but has two drawbacks: first, the KD-Tree is not optimal on the top nodes and, second, increasing the number of computing units will force to use the simplified split heuristic for an increasing number of nodes.

There are two common simplifications for split finding methods. The first consists in using the initial bounding boxes of triangles to evaluate the SAH cost of all split candidates. This implies that the selected split plane is not necessarily the real best choice, because of non-updated triangles limits in the current node. The second consists in only testing a constant number of regularly distributed split planes instead of using real limits of triangles in the current node whereas they are optimal split candidates.

To our knowledge, all the KD-Tree construction algorithms for GPU are built with this kind of approximations. On the CPU side, [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 optimals. [WH06] provides an improved construction algorithm for an optimal KD-Tree that [CKL*10] partially parallelizes on 4-sockets Xeon computer (not a particularly mainstream setting). While increasing construction time, both methods improve traversal time. Nevertheless, they both suffer of the same limit: they are hardly useable in real life situations.

On the traversal side, the classical algorithm, introduced in [HKBv98] is still very efficient and optimized, thanks to early-exit optimization and segmented ray validity handling. The idea is to traverse always the first intersected child to ensure that the best valid intersection found in a visited node is necessarily the final intersection. This allows stopping traversal without visiting any additional node when an intersection is found. However, a dynamic stack is required in order to go back on the last visited node when no valid intersection is found, which is not GPU friendly. ThatâĂŹs why [FS05] introduces a stackless traversal approach for KD-Tree on GPU that keeps all principle of the previous method but by modifying validity range of the ray, restarts from the root node when no intersection is found. Another stackless algorithm was proposed by [PGSS07]. The main idea is to keep ropes that interconnect adjacent nodes. This increases performances by going directly on 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: the memory cost of the tree is typically increased by a factor three.

3. Hybrid KD-Tree Construction

The aim of our building method is to use both the CPU and the GPU, if available, to obtain an optimal KD-Tree, without approximation during the split plane selection. Our first contribution in this area concerns the representation of triangles limits, which are commonly named events. The second contribution concerns a robust and efficient model of task repartition between CPU and GPU.

3.1. Event representation

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

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.*

Without event-type merging	With event-type merging	Acceleration factor
0.534s	0.039s	x13.7

Table 2: *Sorting time acceleration : events representation infuence on NVidia GeForce GTX 460 sorting step for 12M events (i.e.: 2M triangles node.*

[WH06] uses a float for the event's location along with a bool for the start/end type of the event. All the weightiness in the processing of nodes comes from this very simple choice. It adds steps in the ordering predicate when sorting events to ensure that start triangle events are placed before end triangle events in case of equal values and adds a memory access when using events. It also adds 6 bools per triangle in the node structure.

By finding a way to merge events with their types when manipulating the event/type couple, these limits could be overcome. This merging must preserve ordering of events and must limit the KD-Tree sub-optimality to a bare minimum. Concretely, event type only requires one bit. ThatâĂŹs 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 for free that a start event always comes before its corresponding end event, even for aligned triangles.

This encoding allows to directly use high performance libraries providing key/value sort functions, with best performances 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 task repartition is to organize the collaboration between the CPU and the GPU in order to best exploit most of their specificities without being forced to use both of them.

For this, nodes of the KD-Tree are divided into two categories, small nodes and large nodes, depending of 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 ones 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 max depth of the KD-Tree and the total number of triangles in the scene. In our tests, we experimentally fix this threshold to 100000 triangles to obtain the best efficiency.

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 provide the CPU with small nodes as fast as possible.

Note that nodes are sent one by one to the GPU to reduce the maximum memory occupancy. The initial sort is efficiently performed by a key/value sort on primitive types (floats/unsigned), thanks to the event compact representation proposed in section3.1.

The CPU processes small nodes in parallel with one thread per node. This approach is very flexible in term of computing repartition: if a computing unit is unavailable, the CPU will build the entire tree by considering that all the nodes are small nodes. The GPU can also consider that all the nodes are large nodes.

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 the small nodes buffer. They are released when the small nodes buffer is empty and the large nodes thread is over. Note that if a large node canâĂŹt be computed on the GPU, we just have to push this large node into the small nodes buffer and the CPU will process 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 the CPU to finished small nodes processing. These idle times is filled by implementing a large node process for the CPU, like 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 the other using the stackless KD-restart method [FS05]. Both traversal algorithms don't use any paquetization or coherency classification on 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, required by the application. Our task manager can be viewed as a simple counter of remaining rays and a pointer to input (rays)/output (hits) structures. Every tracer can ask the task manager the number of rays to treat. When the job is finished, the task manager stops the tracers.

By using NVIDIA CUDA API [NVI11] zero-copy memory access, ray tracing can be distributed very simply. Indeed, the same input/output pointers and an offset can be passed to all tracers independently of their types. The GPU tracers call a CUDA kernel with these parameters directly, 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. The same function is called, by defining it as *__device__and__host__* function. Load balancing between the CPU and the GPU is a difficult point of 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, for best efficiency, the GPU takes ideally one hundred times more rays than the CPU.

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

5. CUDA implementation

All the GPU part of our hybrid raytracer is implemented with the NVIDIA CUDA API. We chose to use the zero-copy memory to simplify the hybrid part of our raytracer. The KD-tree and the rays/hits buffers are shared by the CPU and the GPU by using this kind of memory. The large node processing, inspired of the [CKL*10] algorithm, can be split in a global initialization step and four node processing steps. Each step is totally adapted for a GPU parallelization.

Step 1: Root initialization (see Figure 1-(1)) is separated into two phases. A first CUDA kernel computes, in parallel for each triangle, the min/max limits on each axis and fills the events value/index by axis. The second step is a call to a key/value sort primitive on events for each axis.

Step 2: Find best split plane (see Figure 1-(2)) for a node . This step takes the buffers of events 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 are then realized. At this point the number of left/right triangles for each split plane candidate generated by events are available. The SAH cost evaluation can now be computed, in parallel, for each split plane candidate. The last thing to do is to find the minimum SAH cost value, by 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 . It works only on events positioned on the best axis plane. The first pass treats all starts events in parallel, and initializes their triangles as left/right triangles in function of the position of the event relatively to the best event. The second pass finishes the task by marking triangles of end events positioned after the best events as shared events, only if it was initialized as left event.

Step 4: Filter geometry and computes new events on each side of the split for the shared triangles (see Figure 1-(4)). This step takes triangle position flags generated by the previous step as input, and classifies events in function of their associated triangles position. Purely left and right triangles are just kept sorted for the next step. Shared triangles events are specially processed. For the split axis, a simplified kernel is launched to clamp shared events with a maximum of the best event value for the left events, and a minimum of the best event value for the right events. For other axis the intersections of shared triangles with the split plane is computed in parallel to obtain new events on each side. At this point, shared events lists for the left and the right side are available.

Step 5: Finalize lists (see Figure 1-(5)) and merges shared events with the left/right side lists , takes the left and right events lists and updated shared events lists of the previous phases. It just consists to merge the left/right events of the current node with the shared left/shared right updated events: first sort the updated shared lists and proceed to a parallel merge of two sorted list for each child of the current node.

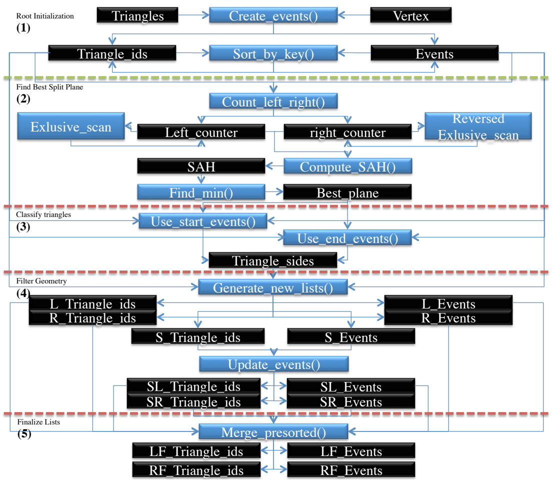

Figure 1: *GPU large node processing implementation details, step by step. Blue cells for CUDA kernels, black cells for data. L=left, R=right, S=shared, F=final.*

Model	[CKL*10]		Our builder		
	1-core CPU^A	32-cores CPU^A	1-core CPU^B	4-cores CPU^B	4-cores CPU^B/GPU
Dragon (871K)	5.5	0.65	2.43	1.42	0.54
Happy (1M)	6.8	0.83	3.2	2.0	0.69
Soda (2M)	/	/	5.2	3.0	1.34

Table 3: *Construction times (in seconds) on some well-known Stanford Computer Graphics Laboratory models, maximum tree depth = 8, NTT = 350000 for all measures. CPUA: Intel Xeon X7550*4 sockets, CPUB: Intel Core i7 920, GPU: NVidia GeForce GTX 460.*

Model	[AL09]	Our tracer	Acceleration
Dragon (871K)	34.5	108.5	x3.15
Happy (1M)	32.5	112.3	x3.45
Soda (2M)	32	52.4	x1.63

Table 4: *Traversal performances for four diffuse rays per pixels (in Mrays.s-1). Results obtained on NVidia GeForce GTX 460.*

6. Results

Event/type merging is the main point of our optimal 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 build times, and allows outperforming the previously available HPC-class hardware performance level.

We chose to confront results with [AL09], a BVH based high performance raytracer. We use the Fermi implementation of the author, available in its public git repository. 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 very high performance when GPU massive parallelism can't be exploited.

References

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 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 k-d tree construction. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 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* (New York, NY, USA, 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 (February 2002), 209–216. 2

[HKBv98] Havran V., Kopal T., Bittner J., Žára J.: Fast robust bsp tree traversal algorithm for ray tracing. *J. Graph. Tools 2* (January 1998), 15–23. 2, 3

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

[NVI11] NVIDIA: The CUDA homepage. http://developer.nvidia.com/category/zone/cuda-zone, 2011. 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