Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

KD-Tree Acceleration Structures for a GPU Raytracer

Tim Foley and Jeremy Sugerman †

Stanford University

Abstract

Modern graphics hardware architectures excel at compute-intensive tasks such as ray-triangle intersection, making them attractive target platforms for raytracing. To date, most GPU-based raytracers have relied upon uniform grid acceleration structures. In contrast, the kd-tree has gained widespread use in CPU-based raytracers and is regarded as the best general-purpose acceleration structure. We demonstrate two kd-tree traversal algorithms suitable for GPU implementation and integrate them into a streaming raytracer. We show that for scenes with many objects at different scales, our kd-tree algorithms are up to 8 times faster than a uniform grid. In addition, we identify load balancing and input data recirculation as two fundamental sources of inefficiency when raytracing on current graphics hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.1 [Computer Graphics]: Raytracing

1. Introduction

The computational demands of raytracing have generated interest in using specialized hardware to accelerate raytracing tasks. It has been demonstrated that raytracing can be achieved in real time on custom hardware [Hal01, SWS02, SWW*04], or by using a supercomputer or cluster of computers [PMS*99, WBS03]. Experiments that used programmable graphics for ray-triangle intersection [CHH02, BFH*04] have also demonstrated that GPUs can outperform CPU implementations.

Purcell et al. [PBMH02] show that the entire raytracing process – camera ray generation through shading – can be implemented on a GPU using a stream programming model. Their work has led to several other GPU raytracer implementations [MFM04, Chr05, KL04] and our work is an extension of their approach.

All of these systems used a uniform grid acceleration data structure. Purcell et al. explain that the uniform grid enables constant-time access to the grid cells, takes advantage of coherence using the blocked memory system of the GPU, and allows for easy iterative traversal via 3D line drawing. It is, however, a suboptimal acceleration structure for scenes with nonuniform distributions of geometry.

The relative performance of different acceleration structures has been widely studied. Havran [Hav00] compares a large number of acceleration structures across a variety of scenes and determines that the kd-tree is the best general-purpose acceleration structure for CPU raytracers. It would seem natural, therefore, to try to use a kd-tree to accelerate GPU raytracing. As we will describe in section 2 though, the standard algorithm for kd-tree traversal relies on a per-ray dynamic stack. Ernst et al. [EVG04] demonstrate that this data structure can be built on the GPU, and implement a stack-based kd-tree traversal. However, their approach requires storage proportional to the maximum stack depth multiplied by the number of rays, which may limit the number of rays that can be traced in parallel. Also, pushing on the stack requires additional render passes with a "scatter" operation.

Our work instead presents kd-tree traversal algorithms *kd-restart* and *kd-backtrack* that run without a stack. We show that these new algorithms maintain the expected performance of kd-tree traversal. We also present a GPU-based raytracer that incorporates these algorithms and demonstrate that, as on CPUs, they outperform uniform grid acceleration structures with scenes of sufficient complexity. Finally, we

† {tfoley, yoel}@graphics.stanford.edu

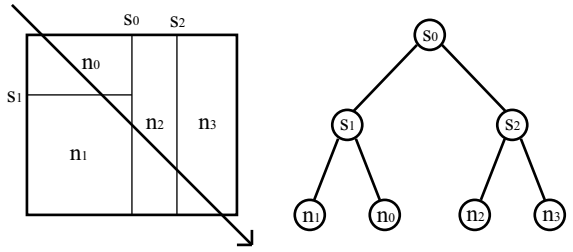

Figure 1: *Left: A two-dimensional kd-tree. Internal nodes are labeled next to their split planes and leaf nodes are labeled inside their volume. Right: A graph representation of the same kd-tree.*

discuss the major factors determining performance and discuss how future hardware improvements could affect GPU kd-tree traversal.

2. The KD-Tree Algorithm

To better illustrate the variations to the kd-tree traversal algorithm, we will utilize the simple two-dimensional kd-tree and input ray shown in Figure 1. Each node in the tree represents an axis-aligned rectangular region of space, and each internal node is annotated with an axis-aligned plane that separates the regions of its two children.

2.1. The Standard Approach

A typical kd-tree traversal algorithm for raytracing, taken from [PH04], is shown in Figure 2. The algorithm takes as input a tree and a ray, and searches for the first primitive in the tree that is intersected by the ray. The tree is traversed starting at the root, and a stack is used as a priority-ordered list of nodes left to visit. Each node on the stack is closer to the ray origin than all nodes below it, and the node currently being traversed is closer than all nodes on the stack. A $(tmin, tmax)$ range limits the part of the ray under consideration to that which intersects the current node.

When an internal node is encountered during the traversal, the $(tmin, tmax)$ range of the ray is classified with respect to the splitting plane of the node. If the range lies entirely to one side of the plane, the traversal simply moves to the appropriate child. If, instead, the range straddles the plane then traversal will continue to the first child hit by the ray, while the second child is pushed onto the stack along with its appropriate $(tmin, tmax)$ range. In this way the traversal proceeds down the tree, occasionally pushing items onto the stack, until a leaf node is reached.

If the ray intersects one of the primitives in the leaf within the $(tmin, tmax)$ range, then the closest intersection in the leaf is guaranteed to be the first intersection along the ray, and the traversal terminates and yields this result. If no intersection is found then we pop a work item – consisting of a

```
kd-search( tree, ray )
   (global-tmin, global-tmax) = intersect( tree.bounds, ray )
   search-node( tree.root, ray, global-tmin, global-tmax )

search-node( node, ray, tmin, tmax )
   if( node.is-leaf )
      search-leaf( node, ray, tmin, tmax )
   else
      search-split( node, ray, tmin, tmax )

search-split( split, ray, tmin, tmax )
   a = split.axis
   thit = ( split.value - ray.origin[a] ) / ray.direction[a]
   (first, second) = order( ray.direction[a], split.left, split.right )

   if( thit >= tmax or thit < 0 )
      search-node( first, ray, tmin, tmax )
   else if( thit <= tmin )
      search-node( second, ray, tmin, tmax )
   else
      stack.push( second, thit, tmax )
      search-node( first, ray, tmin, thit )

search-leaf( leaf, ray, tmin, tmax )
   // search for a hit in this leaf
   if( found-hit and hit.t < tmax )
      succeed( hit )
   else
      continue-search( leaf, ray, tmin, tmax )

continue-search( leaf, ray, tmin, tmax )
   if( stack.is-empty )
      fail()
   else
      (n, tmin, tmax) = stack.pop()
      search-node( n, ray, tmin, tmax )
```

Figure 2: *Standard kd-tree traversal pseudocode. Note that all function calls are tail calls, and the algorithm can thus be implemented iteratively.*

node and a $(tmin, tmax)$ range – from the stack and continue searching. If the stack is empty then there is no intersection along the ray and the search terminates.

While the worst case performance of this algorithm is $O(n)$ in the number of leaf nodes n, its expected cost for real scenes is $O(log(n))$. As illustrated in Figure 1, in the worst case a ray may visit a number of nodes linear in the size of the tree. In practice, however, it is expected that most rays will find an intersection within one of the first leaf nodes visited [Hav00], so the expected performance is proportional to the height of the tree, which will typically be logarithmic.

2.2. The KD-Restart Algorithm

The *kd-restart* traversal algorithm modifies the standard kd-tree traversal to eliminate all stack operations. If we eliminate the *push* operation from the *search-split* routine in figure 2, then the resulting traversal will proceed directly to the first leaf pierced by the ray. We can observe that so modified, the traversal will only change the value of *tmax* in those cases where an item would have been pushed onto the stack. In fact, the new value of *tmax* in such a case will be exactly the value pushed onto the stack as *tmin*. This implies that when we reach a leaf, the value of *tmax* is either the global *tmax* value, or it is exactly the *tmin* value at which the ray enters the next leaf. We take advantage of this fact by modifying the *continue-search* routine:

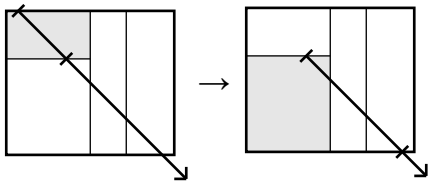

Figure 3: *After failing to find an intersection in a leaf node, kd-restart advances the* $(tmin,tmax)$ *range forward. Note that the modified range now starts in the next leaf to be traversed.*

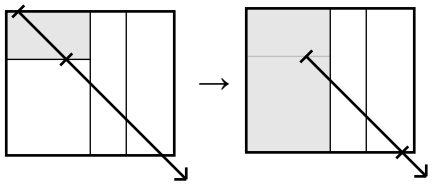

Figure 4: *After searching a leaf node, kd-backtrack resumes the search at the first ancestor that intersects the modified* $(tmin,tmax)$ *range.*

```
continue-search( leaf, ray, tmin, tmax )
   if( tmax == global-tmax )
      fail()
   else
      tmin = tmax
      tmax = global-tmax
      search-node( tree.root, ray, tmin, tmax )
```

When the *kd-restart* traversal reaches a leaf node and fails to find a hit, it simply restarts the search at the root of the tree with the value of *tmin* advanced to the end of the leaf. As illustrated in Figure 3, the first leaf intersected by the modified range is the next leaf that needs to be traversed. By repeating this process we will visit all the leaves along the ray in the same order as the standard kd-tree traversal. However, as each leaf node is reached by a traversal starting at the root of the tree, the cost of a traversal that visits m leaf nodes is $O(m \cdot h)$ where h is the height of the tree. In a balanced tree this yields a worst-case cost of $O(n \cdot log(n))$. If we assume, however, that the average number of leaf nodes visited is bounded by a small constant in practice, then our expected cost will still be $O(log(n))$.

2.3. The KD-Backtrack Algorithm

The *kd-backtrack* algorithm modifies *kd-restart* to maintain linear worst-case bounds at the cost of additional per-node storage. We first observe that the nodes that are pushed on the stack in the traditional algorithm are always the "other" (that is, second) child of one of the current node's ancestors. Thus it should be possible to reach the parent of the node atop the stack by following a chain of parent links (which we can store in the nodes of the tree) from the current node.

If we again employ the tactic of advancing *tmin* to the end of the last leaf visited, then we will be able to recognize the appropriate parent as the closest ancestor that has a nonempty intersection with the remaining $(tmin,tmax)$ range. Figure 4 demonstrates this effect. We can perform this intersection test using an axis-aligned bounding box stored along with every internal node. We implement this approach with a modified *continue-search* routine:

```
continue-search( leaf, ray, tmin, tmax )
   if( tmax == global-tmax )
      fail()
   else
      tmin = tmax
      tmax = global-tmax
      backtrack( leaf.parent, ray, tmin, tmax )

backtrack( split, ray, tmin, tmax )
   (t0,t1) = intersect( split.bounds, ray, tmin, tmax )
   if( no-intersection )
      backtrack( split.parent, ray, tmin, tmax )
   else
      search( split, ray, t0, t1 )
```

The *kd-backtrack* algorithm increases per-node storage costs – it requires bounding boxes to be stored with internal nodes, as well as parent links in all nodes. However, unlike *kd-restart*, it preserves the worst-case asymptotic time complexity of the standard traversal algorithm. To see this, we observe that the backtracking step will not visit any node that was not previously visited by the downward search. Furthermore, a given node will be visited at most two times by the backtracking step; once coming from the left child, and once coming from the right. Thus *kd-backtrack* is at most a factor of three slower than the standard algorithm.

3. Implementation

We have implemented our kd-tree traversal algorithms inside of a streaming GPU raytracer modeled after [PBMH02]. Our raytracer is built on the Brook for GPUs [BFH*04] environment for stream processing on graphics hardware, and includes support for four acceleration structures:

- Brute Force – The simplest intersection scheme, similar in approach to that of [CHH02]. It intersects an entire stream of rays with a single triangle at a time, passed in using constant arguments. On an ATI Radeon X800 XT PE [ATI04] we have measured this kernel to achieve nearly 350 million ray-triangle intersection tests per second.
- Uniform Grid – A uniform grid intersection algorithm similar to that presented in [PBMH02] Because the Brook system does not support storing streams as 3D textures we pack the voxel data into a large 2D texture and use a small number of instructions to index it. Because of limited arithmetic precision on the ATI hardware we used for testing, this effectively limits our scenes to 2^{17} voxels maximum.
- KD-Restart – A GPU implementation of the *kd-restart* algorithm.
- KD-Backtrack – A GPU implementation of the *kd-backtrack* algorithm.

```
struct Leaf { // a leaf node in the kd-tree
   float triangleCount;
   short2 firstTriangle;
   short2 parentPointer;
}

struct Split { // an internal node in the kd-tree
   float splitValue;
   short2 leftChildPointer;
   short2 rightChildPointer;
   // sign bits store child type (leaf/split) and split axis
}

struct ExtendedSplit { // further data for kd-backtrack
   float3 boundingBoxMin;
   float3 boundingBoxMax;
   float2 parentPointer;
}
```

Figure 5: *Data types used to store the kd-tree in GPU memory.*

```
struct TraversalState {
   float2 nodePointer;
   float tmin;
   float tmax;
   // sign bits store node type (leaf, split)
   // and ray state (traverse, intersect, done)
}

struct IntersectState {
   float2 triangleIndex;
   float triangleCount;
   float tmax;
}

struct HitState {
   float2 bestTriangleIndex;
   float thit;
   float global-tmax;
}
```

Figure 6: *Data types used to store per-ray traversal state.*

3.1. KD-Tree Memory Use

In order to make the most efficient possible use of the limited GPU memory, and to minimize the bandwidth cost of our kernels, our kd-tree algorithms store the tree and per-ray state using several packed data structures, as listed in Figures 5 and 6. As implemented, the base kd-tree structures use 96 bits of storage per node, while typical CPU implementations require only 64 bits per node [PH04]. The parent pointer in the *Leaf* structure, however, is only needed for *kd-backtrack* and could be eliminated in a restart-only implementation. In addition, by using the standard technique of storing each left child at a known offset from its parent node, we could eliminate one pointer from the *Split* structure. In this way, a *kd-restart* implementation could store the tree as efficiently as most CPU kd-tree implementations. The *ExtendedSplit* structure required by the *kd-backtrack* algorithm, however, incurs an additional 256 bits of storage costs per internal node.

3.2. KD-Tree Construction

Both our uniform grid and kd-tree data structures are built offline on the CPU. Our kd-tree construction algorithm is based on [PH04]. This algorithm uses a surface-area heuristic along with approximate cost information as introduced by [MB90] and improved by [HB02]. Even though intersection is more costly than traversal in our implementation, we have found that using a cost model which regards traversal as being 8 times more costly than intersection generates good trees for our system. This has the effect of keeping our trees relatively shallow and increasing the number of primitives in each leaf node. These effects serve to increase coherence between rays and may be responsible for improving performance.

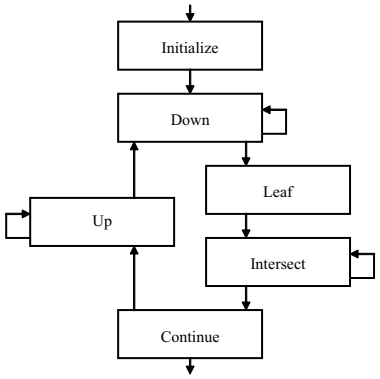

Figure 7: *Dataflow between kernels used in the kd-backtrack implementation. The kd-restart implementation eliminates the Up kernel.*

3.3. KD-Tree Kernels

The two kd-tree implementations are based on a set of six Brook kernels. Figure 7 illustrates how these kernels are sequenced for the *kd-backtrack* algorithm. The kernels used are:

- Initialize – Corresponds to the routine *kd-search*. This kernel finds the global $(tmin, tmax)$ range for the ray and initializes its traversal.
- Down – Corresponds to the routine *search-split*. This kernel find the first child (left or right) of the current node pierced by the ray and traverses to it.
- Leaf – Corresponds to a call to routine *search-leaf*. This kernel is used to take a ray that has been traversing and prepare it to intersect triangles.
- Intersect – Corresponds to the body of routine *search-leaf*. This kernel iteratively processes the triangles within the current leaf node.
- Continue – Corresponds to the body of routine *continue-search*. This kernel determines whether a hit has been successfully found and either terminates the query or continues it (by restarting or initiating backtrack, as appropriate).
- Up – Corresponds to the *backtrack* routine. This kernel advances a ray query up the tree until the new $(tmin, tmax)$ range intersects some node's bounding box.

3.4. Kernel Masking and Scheduling

As mentioned in [PBMH02], achieving high performance without branching requires careful scheduling of kernels and masking of inactive elements. Similar to their approach, we run our kernels in multiple rendering passes and use the hardware z-buffer and early-z culling to avoid spending computation on inactive rays. We select the kernel to run in each pass using the load balancing strategy from [Del88]. We maintain an estimated count of rays in each state and run a kernel until the estimate for the next kernel is more than twice the estimate for the current one. This heuristic is meant to reduce the overhead of switching states (which requires refilling the entire z-buffer, switching kernels, and switching inputs) while still operating on large batches of rays. We use hardware occlusion queries to gather our estimates of rays in each state.

Recent GPUs have started to offer data-dependent per-fragment branching and looping instructions [NVI04], which reduce the need for multiple passes and sophisticated scheduling. However, their effect on our kd-tree algorithms would be minimal. Each of our kernels corresponds roughly to the body of a loop in a branching kernel. As a result, the steps run on a given ray are the same in our implementation as they would be in a branching version. Additionally, Wenzel [Wen05] reports that the performance of current branching implementations is not high and Buck [Buc05] discusses stability problems and the need for high locality between branches taken by nearby fragments. We performed preliminary studies on a branching version of our code and had similar experiences. Therefore, we restricted our implementation to a kernel-level masking and scheduling approach.

4. Results

We tested our raytracer on a 256MB ATI X800 XT PE [ATI04] running the Catalyst 4.10 drivers. We evaluated our raytracer's behavior while producing 512x512 images of four scenes:

- Cornell Box - 32 triangles, 2x2x2 uniform grid. An extremely simple scene.
- Stanford Bunny - 69451 triangles, 50x50x50 uniform grid. A scene with a single object and a fairly regular distribution of triangles.
- Robots - 71708 triangles, 50x50x50 uniform grid. A scene from the BART [LAM00] suite that exhibits the "teapot in a stadium" scenario. There are many detailed robots in a larger, less detailed, city.
- Kitchen - 110561 triangles, 50x50x50 uniform grid. Another scene from BART that models a kitchen scene with a variety of objects of different sizes and complexities.

The views rendered for each of our test scenes are presented in Figure 8.

	Brute	Grid	Restart	Backtrack
Cornell Box	23	63	80	84
Bunny	4620	357	701	690
Robots	4770	8344	968	946
Kitchen	7350	2687	992	857

Table 1: *Elapsed time (milliseconds) for the primary ray casting of the scenes in Figure 8 with the different algorithms.*

	Down/Up	Intersect	Trans.
Robots/Restart	31.68M/0M	6.84M	2.48M
Robots/Backtrack	13.49M/9.80M	6.84M	2.48M
Kitchen/Restart	21.80M/0M	5.91M	2.13M
Kitchen/Backtrack	10.86M/7.78M	5.91M	2.13M

Table 2: *The total number of times primary rays were in each state of the two kd-tree algorithms for different scenes. Because of masking, these counts do not correspond to the number of kernel calls. The Transition counts reflect both the Leaf and Continue kernels.*

4.1. Rendering Performance Tests

Table 1 shows the time spent in the scene-intersection portion of rendering the test scenes at a resolution of 512x512. As expected, the brute force approach is linear in the number of triangles. The uniform grid and two kd-tree approaches behave as expected. The uniform grid handles the regularity of the Stanford Bunny extremely efficiently (an effective rate of over 700 thousand rays per second), but its performance on the Robots scene is more than 20 times slower, despite the similar triangle counts of these scenes. These results confirm the findings of similar studies on the CPU. Grids work better for scenes with uniform distributions of similarly-sized objects, while kd-trees work better when the scene contains a non-uniform distribution of objects of different sizes. Note that the absolute performance of our kd-tree algorithms is roughly 250-300 thousand rays per second for the complex Robots and Kitchen scenes.

For these scenes, the performance of the *kd-restart* and *kd-backtrack* algorithms was very similar. In order to understand the differences in behavior, we use occlusion queries to count how often rays were in each state of the two algorithms for the Robots and Kitchen scenes. These results are summarized in Table 2. As expected, both algorithms visited the same number of leaves and intersected the same number of triangles. The *kd-backtrack* algorithm merely optimizes how it reaches leaves by backtracking only as far up the tree as necessary.

These measurements also allow us to estimate how a

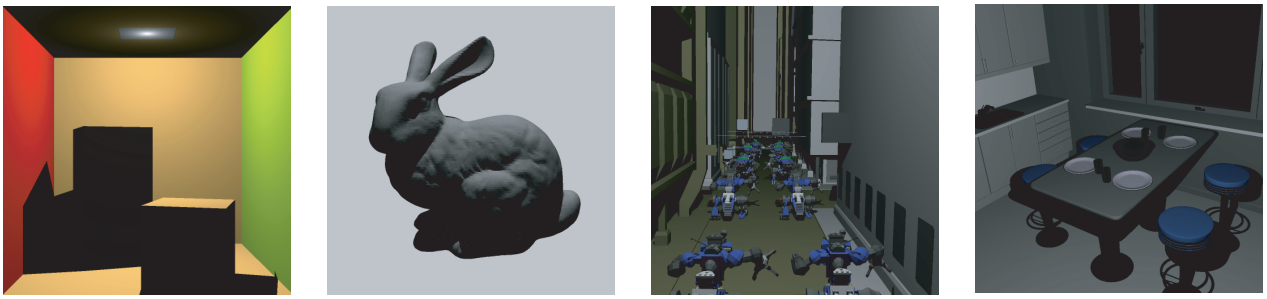

Figure 8: *Four scenes used to test our GPU raytracer. From left to right: The Cornell Box, The Stanford Bunny, BART Robots, and BART Kitchen.*

	Instr.	Floats In/Out	Dependent Fetches	GB/s
Brute Intersect	37	12/4	0	20.7
Grid Traverse	57	26/16	2	10.8
Grid Intersect	83	36/8	16	16.6
KD Down(*)	66	15/4	3	15.7
KD Up	36	20/4	8	24.5
KD Intersect	72	33/8	13	18.5

Table 3: *Characteristics of the key kernels. The KD Down kernel is compute-bound. Despite attaining different bandwidths, the other kernels all turn out to be at or nearly bandwidth-bound.*

stack-based algorithm would compare to our stackless implementations. The *kd-backtrack*'s Down count in Table 2 is exactly the number of search steps that the stack-based approach would have required. The Up steps represent the extra work required by our backtracking algorithm. Even with this work, there is less than a two-fold increase in the total traversal work. Similarly, we find that the extra work performed by *kd-restart* is within a factor of three of the traversal work performed by a stack-based kd-tree traversal algorithm. This resonates with the belief that in the average case, each kd-tree ray traversal terminates after visiting only a small number of leaf nodes (and thus only incurs only a small number of restarts or backtracks).

4.2. Kernel Performance Tests

We also evaluated the raw performance of the key kernels used by our acceleration structures. In general, a kernel's performance is limited by one of two factors: the rate at which the hardware can execute math instructions (compute) or the rate at which the hardware can fetch the required texture data (bandwidth). In the first case we say the kernel is compute-bound; in the second, we say that it is bandwidth-limited. Current GPUs require significant computation to cover the cost of texture fetches [FSH04] so we expected and found that bandwidth is the critical resource.

We took the rays and triangles from the Kitchen scene and measured the elapsed time to run 500 invocations of each kernel (without any z-buffer culling or load balancing). Using the number of floats fetched from textures and written to outputs we computed the effective bandwidth. Table 3 shows the results.

We determined which kernels were bandwidth-limited by timing fetch-only versions. These kernels were stripped down to perform the same number of fetches and writes, but only enough math to prevent the driver from eliminating the fetches. Where the fetch-only versions exhibited the same performance as the full versions, we concluded that the full versions were bandwidth-bound or near the crossover point. Only the kd-tree Down kernel proved to be compute-bound.

There are two reasons that our bandwidth-bound kernels all reach different effective bandwidths. The first is that the ATI drivers / hardware penalize the use of multiple outputs (each output is a float4 vector). There is a small falloff in achievable output bandwidth with a second output which affects the grid and kd-tree Intersect kernels. There is a much larger falloff with four outputs which impacts the uniform grid Traverse kernel.

The second determining factor is the method and pattern of accesses to input textures. [FSH04] reports that fetching texture data already in the texture cache is nearly twice as fast as sequential streaming accesses. The brute force Intersect kernel is a purely streaming kernel and its input bandwidth (three quarters of its total bandwidth) is in sync with the reported 15.6 GB/s rate for sequential cache accesses.

Additionally, most of our kernels have data-dependent fetches – values read from one input texture are used to compute the addresses of values fetched from others. Dependent fetches allow completely arbitrary access patterns and their performance is determined heavily by their coherence. Patterns where nearby fragments load from the same or overlapping addresses perform well, even better than sequential

accesses. Texture units in hardware are heavily optimized for this sampling pattern. On the other hand, random accesses perform very poorly.

The kd-tree Up kernel reaches the highest effective bandwidth because it produces exponentially increasing coherence as it iterates. Each step of the Up kernel moves rays closer to the root of the kd-tree and each level has roughly half as many nodes as the one beneath it. That means that even a random distribution of rays will be focused into more and more coherence as it progresses. The kd-tree Up kernel and both the kd-tree and uniform grid Intersect kernels also receive coherence as a result of nearby rays tracing similar paths through the scene and being tested against similar triangles. This latter form of coherence, however, would be reduced tracing secondary rays like shadow rays while the focusing coherence would remain.

The kd-tree Down kernel is the one compute-bound kernel. On the CPU the traversal algorithm has been heavily optimized and needs much less computation than our equivalent kernel. In contrast, the Down kernel cannot be optimized further due to existing GPU limitations. The data it fetches is heavily packed (to conserve memory and bandwidth) and unpacking requires computation. Current GPUs lack integer and bitwise operations which exacerbates the cost of unpacking. Additionally, predication is the only conditional primitive available on our hardware. This causes us to incur the cost of all paths through the branches in the algorithm. Finally, the current ATI hardware's poor numerical precision (only 16 bits of mantissa) forced us to add an extra conditional to eliminate numerical instability when *tmin* is very close to a splitting plane – a situation that our modified kd-tree traversal algorithms make common.

5. Conclusions and Future Work

We have described a modified kd-tree traversal algorithm that eliminates the per-ray stack. While the removal of the stack was motivated by the limitations of current GPUs, it also reduces the working set size needed to trace rays through a kd-tree. This change allows for more rays to be traced in parallel, which we believe could benefit a number of architectures. We compare our streaming kd-tree implementation with a streaming uniform grid and see the same relative strengths as their CPU counterparts: the grid is efficient at uniformly populated scenes but is not as fast as the kd-tree in complex scenes with varying levels of detail.

Despite the improvements we have introduced to our raytracer, our performance is only a few hundred thousand rays per second while the fastest CPU implementations report rates up to a few million rays per second [BWS03]. Our implementation is not as heavily optimized as the best CPU systems, but this still seems a surprising disparity given our GPU's many-fold advantage in both peak computational rate and memory bandwidth. We see two fundamental issues limiting the performance of our raytracer – the inefficiency of load balancing and the costs of data recirculation. We believe these issues are central for any highly parallel raytracer.

We depend on compute masking and careful scheduling of kernels to run our algorithms on the extremely wide SIMD architecture of current fragment processors. Our current load balancing techniques get very low utilization of the hardware, and we believe that this incurs more than a factor of five slowdown. Performance is lost both to the cost of switching kernels, and to the low efficiency of the hardware when shading small numbers of fragments. Conditional branching support could allow for better masking and scheduling of fragments, and it will be instructive to retry branching versions of our kernels as hardware offerings mature. Unfortunately, this approach puts the burden of efficiency on the branching hardware, and that poses a challenge to the SIMD organization of current architectures.

The pure streaming approach to GPU raytracing devotes a large portion of the available bandwidth to recirculating per-ray data between passes. Each kernel reloads e.g. the ray origin and direction and fetches the ray state written by the previous pass. This data competes with dependent fetches in kernels that are largely bandwidth bound. In a conventional threaded raytracer, this per-ray data would be loaded into registers or local cache and would stay there throughout traversal. Per-fragment branching would allow the same approach to be applied on GPUs. Alternatively, if peak performance could be reached with a smaller number of fragments, a local memory to cache reused input data and iteration state across kernel invocations could have the same effect.

Though streaming raytracers have not yet tapped the full performance of GPUs, kd-trees bring increased scalability and demonstrate that the potential for improvement still exists. We are hopeful that our kd-tree traversal algorithms will encourage further exploration of data structures and algorithms for streaming raytracing, and that future generations of hardware will allow for more efficient implementation of our algorithms.

6. Acknowledgments

We would like to thank Tim Purcell for providing access to his GPU raytracer and Matt Pharr and Greg Humphreys for the pbrt raytracer. Mike Houston provided a wealth of assistance with our performance analysis, and Pat Hanrahan gave invaluable feedback during the writing process.

Support for this research was provided by the Rambus Stanford Graduate Fellowship, DARPA Polymorphous Computing Architectures Project (contract F29601-03-0117-P00004), and the DARPA Smart Memories Project (contract MDA904-98-R-S855).

References

[ATI04] ATI: Radeon X800 product site, 2004. http://www.ati.com/products/radeonx800. 3, 5

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of ACM SIGGRAPH 2004* (2004). 1, 3

[Buc05] BUCK I.: *Stream Computing on Graphics Hardware*. Ph.D. thesis, Stanford University, Mar. 2005. 5

[BWS03] BENTHIN C., WALD I., SLUSALLEK P.: A scalable approach to interactive global illumination. In *Proceedings of Eurographics* (2003), vol. 22, pp. 621–630. 7

[CHH02] CARR N. A., HALL J. D., HART J. C.: *The Ray Engine*. Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinois, 2002. 1, 3

[Chr05] CHRISTEN M.: *Ray Tracing on GPU*. Diploma thesis, University of Applied Sciences Basel, Switzerland, 2005. 1

[Del88] DELANY H. C.: Ray tracing on a connection machine. In *Proceedings of the 1988 International Conference on Supercomputing* (1988), pp. 659–667. 5

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization 2004* (2004), pp. 255–262. 1

[FSH04] FATAHALIAN K., SUGERMAN J., HANRAHAN P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004* (Aug. 2004), p. page range goes here. 6

[Hal01] HALL D.: The AR350: Today's ray trace rendering processor. 2001 SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session 1, 2001. http://graphicshardware.org/previous/www_2001/presentations/Hot3D_Daniel_Hall.pdf. 1

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 1, 2

[HB02] HAVRAN V., BITTNER J.: On improving kd-trees for ray shooting. In *Proceedings of WSCG'2002 conference* (2002), pp. 209–217. 4

[KL04] KARLSSON F., LJUNGSTEDT C. J.: *Ray Tracing Fully Implemented on Programmable Graphics Hardware*. M.S. thesis, Chalmers University of Technology Göteborg, Sweden, 2004. 1

[LAM00] LEXT J., ASSARSSON U., MOELLER T.: *BART: A Benchmark for Animated Ray Tracing*. Tech. rep., Chalmers University of Technology, Goeteborg, Sweden, May 2000. http://www.ce.chalmers.se/BART/. 5

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (1990), 153–166. 4

[MFM04] MORENO-FORTUNY G., MCCOOL M.: Unified stream processing raytracer. Poster at GP^2: The ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster, 2004. http://www.cgl.uwaterloo.ca/ gmoreno/streamray.html. 1

[NVI04] NVIDIA: Geforce 6 series technical specifications, 2004. http://www.nvidia.com/object/geforce6_techspecs.html. 5

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Trans. Graph.* (2002), 703–712. 1, 3, 5

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004. 2, 4

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics* (New York, NY, USA, 1999), ACM Press, pp. 119–126. 1

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: SaarCOR – a hardware architecture for realtime ray-tracing. In *Proceedings of EUROGRAPHICS Workshop on Graphics Hardware* (2002). available at http://graphics.cs.uni-sb.de/Publications. 1

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware* (2004), pp. 95–106. 1

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003). 1

[Wen05] WENZEL C.: Far Cry and DirectX. Game Developers Conference, 2005. http://download.nvidia.com/developer/presentations/2005/GDC/Direct3D_Day/D3DTutorial08_FarCryAndDX9.pdf. 5