Parallel Divide and Conquer Ray Tracing

Srinath Ravichandran* IIIT - Hyderabad

P.J.Narayanan† IIIT - Hyderabad

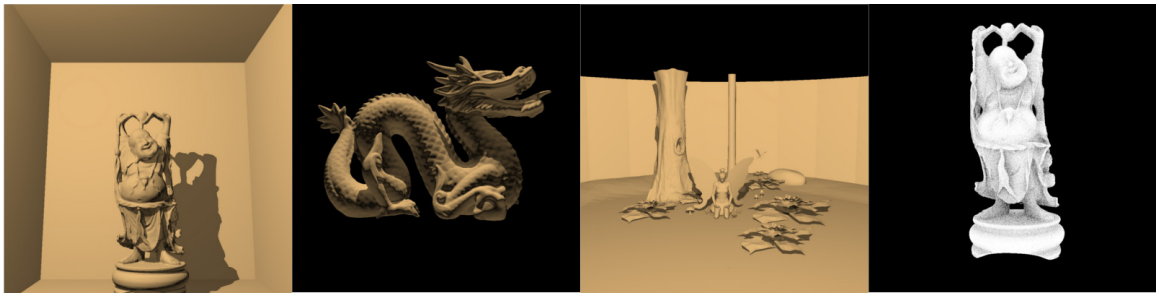

Figure 1: *Buddha (1.08M, 315 ms), Dragon (871K, 283 ms) and Fairy (174K, 276ms) models rendered with shadows using our approach. Right: Buddha model with 8 ambient occlusion rays per pixel (410 ms). Resolution* 1024×1024.

Abstract

Divide and Conquer Ray Tracing (DACRT) is a recent technique which constructs no explicit acceleration structure. It creates and traverses an implicit hierarchy in a depth-first fashion recursively and is suited for dynamic scenes that change constantly. In this paper, we present a parallel version of DACRT that runs entirely on the GPU, which exploits efficient primitives like sort and reduce. Our approach suits the GPU well, with a low memory footprint. Our implementation outperforms the serial CPU algorithm for both primary and secondary ray passes. We show good performance on primary pass and on advanced effects.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: Raytracing, Graphics Processors, Divide and Conquer, Parallel Processing

1 Introduction

Ray tracing is important to generate photo-realistic images. Efficient ray-tracing uses structures such as BVH, k-d trees, grids, for efficiency. The construction is expensive and acceleration structures have high memory requirements. Several methods to construct acceleration structure in parallel on CPUs and GPUs have appeared recently. For general dynamic scenes, the acceleration structure has to be reconstructed every frame. Divide and conquer ray tracing (DACRT) aims at resolving these problems [Mora 2011; Áfra 2012; Nabata et al. 2013]. DACRT traces rays through a 3D scene without constructing an explicit acceleration structure. It takes an array of scene primitives, its bounding box, and an array of rays as input. It then reduces the problem by recursively subdividing the space and partitioning the elements of the ray and primitive lists that satisfy a basic intersection condition. When the problem size is small enough, it uses an exhaustive procedure to compute intersections between all rays and primitives.

In this paper, we present the first fully parallel DACRT algorithm. We use a breadth-first traversal of the space to extract parallelism and exploit parallelism at all stages of the algorithm on the GPU. We also exploit efficient primitives on the GPU for several key steps to get maximum performance. We achieve good speed up compared to the CPU DACRT method. Figure 1 gives example images with timing for different models and effects.

2 Parallel DACRT

The pseudo code for our algorithm is given in Algorithm 1. A *node* represents a node in the implicit hierarchy. It contains a number of triangles and an enclosing AABB, along with a number of rays that intersect the AABB. A *child-node* is created from a parent node by splitting its AABB according to some condition, and contains a subset of its parent's triangles and rays. A *level* represents the group of nodes at a fixed depth from the root of the implicit hierarchy. A *pivot* is a two component vector that stores the start and end points of a group of elements belonging to a node within a linear list. Every ray stores the distance *mints* to the nearest intersected object indicated by *hitid*. *Global mints* and *Global hitid* arrays store parameter information for all input rays. *Terminal-nodes* have triangle or ray count below a fixed threshold. *Terminal-node buffer* represents a buffer in which all terminal nodes are stored temporarily before processing. The buffer also contains a separate *mints* and *hitid* arrays to store temporary ray parameters.

The Parallel DACRT algorithm starts with a linear list of rays and triangles along with the scene bounding box as shown in Figure 2. The algorithm iteratively constructs an implicit hierarchical structure over the list of triangles and traverses it level by level at the same time. The data is always kept in linear lists though referred to as *nodes* and *levels*. Nodes are implicitly defined over a linear list (*level*) using pivots that indicate the range of elements belonging to it. Nodes have pivots for both triangle and ray lists. The implicit hierarchy is created by splitting nodes into children. The splitting scheme defines the type of hierarchy constructed. Spatial splitting of nodes results in a k-d tree while splitting of objects results in a BVH. We follow a spatial splitting scheme which splits a node's

*e-mail:srinath.ravichandran@research.iiit.ac.in

†e-mail:pjn@iiit.ac.in

The definitive version was published in SA'13 November 19-22, 2013. http://dx.doi.org/10.1145/2508363.2524698

Algorithm 1 Parallel DACRT

```
1: procedure ParallelDacrt(TriangleArray △, RayArray ↑,
   Scene AABB □, Global mints array, Global hitid array)
2:     Initialize level with root info
3:     while true do
4:         Split all current level nodes into child nodes in parallel
5:         Perform Triangle & Ray Filtering in parallel
6:         Calculate child-node pivots; mark terminal-nodes
7:         if (terminal node buffer almost full) then
8:             Process all terminal-nodes
9:             Update global mints and global hitid
10:        end if
11:        Compute next level node sizes in parallel
12:        Gather next level triangle and ray index data in parallel
13:        Copy terminal-nodes to buffer in parallel
14:        Re-index next level pivots in parallel
15:        Create next level and assign it to current level
16:        Free memory for current level
17:    end while
18:    if (terminal-node buffer not empty) then
19:        Process all remaining terminal nodes
20:        Update global mints and global hitid
21:    end if
22: end procedure
```

AABB into two child AABBs at the midpoint of the largest extent dimension. Once a node is split, the filtering operation determines the triangles and rays that intersect the child AABBs. Ray filtering works similar to a breadth first ray tracing where groups of rays are traced together down the implicit hierarchy. This scheme prevents the use of early ray termination as construction and building are performed together. However, the ray and triangle filtering offer fine-grained parallelism as every ray and triangle within every node can be handled independently. We exploit this to accelerate the process as a whole.

2.1 Parallel Filtering

The pseudo code for parallel filtering operation is given in Algorithm 2. *Element list* corresponds to either the ray list or triangle list and *Element id list* corresponds to respective id list.The filtering operation defined in Line 5 in Algorithm 1 is a per element intersection test with an AABB. We refer to the result of the filtering operation as a *status code* (Figure 3). The *status codes* returned by the filtering operations are used to rearrange the elements of the list so that elements with the same code occur contiguously as segments within the list (Figure 4). The *segments* correspond to nodes and are identified uniquely using *pivots*.

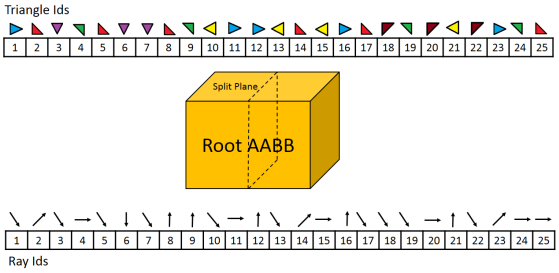

Figure 2: *Implicit root node is defined over the input triangle and ray index lists. Each triangle and ray is handled by a thread to determine if it intersects the child AABBs afer split.*

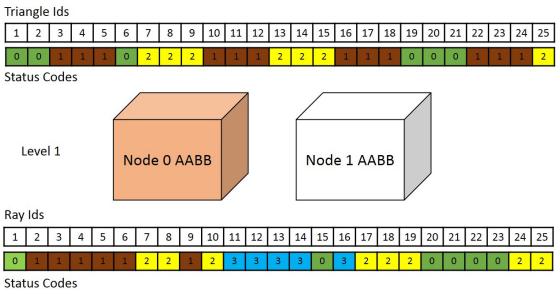

Figure 3: *Child AABBs created after splitting. Parallel candidate tests on triangles and rays yield the status codes as shown. Elements have their status codes coloured to indicate left (green), both (brown), right (yellow) or none(blue)*

Consider a level (`Root` in Figure 2) in the hierarchy containing n nodes each with t triangles and r rays that needs filtering. When a parent node is split, its triangles can be in one, both, or none of the child nodes. A ray of the parent's AABB can intersect one, both, or no children's AABB. Hence, the output of the candidate tests for triangles can be *left*, *right* or *both*, encoded for each triangle as 1, 3, or 2, respectively. For rays, the possible results are *left*, *right*, *both* or *none*, encoded respectively as 1, 3, 2, and 4 (Figure 3). The status code is encoded as an unsigned integer with 30-bits for the node number and right-most 2 bits for the intersection result. We then sort with the status code as the key and the element id as the value. The elements that straddle both children will be between those that intersect the left or right child after the sort (Figure 4). We can then extract the pivot ranges for the left and right children of each node easily from this representation. The left pivot's range would cover elements with codes *left* and *both* and right pivot's range would cover *both* and *right* (Figure 4). A parallel reduce operation on the status codes would yield values that are multiples of their respective status codes. Dividing each value by its status code would return the number of elements for each node considered (Figure 5).

After parallel filtering, a kernel marks all nodes that have low triangle ($< \alpha$) or ray ($< \beta$) counts as terminal-nodes (Figure 6). We then compute the sizes required for the next level nodes to store the ids. We allocate enough memory for the next level nodes and a parallel gather operation copies nodes into their respective locations.

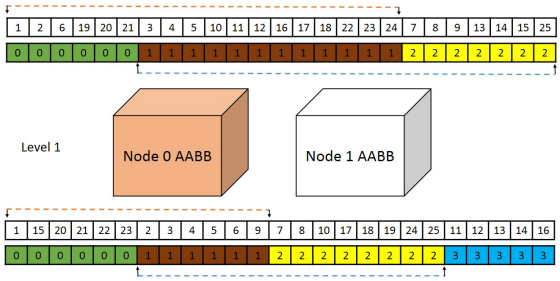

Figure 4: *Sort-by-key rearranges elements of the two child nodes maintaining pivot property. The pivot ranges are marked with coloured dashed lines. Pivot ranges can substantially overlap between sibling nodes at any level.*

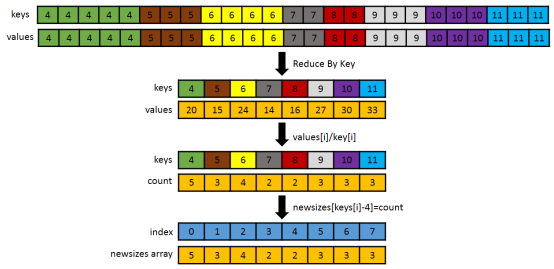

Figure 5: *Parallel size calculation for child nodes at a level*

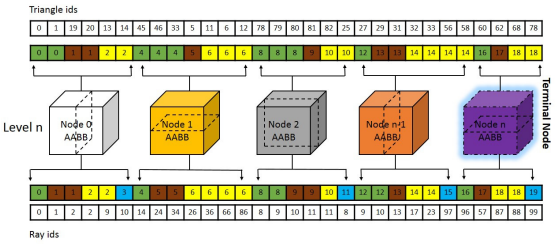

Figure 6: *Nodes at level n. All nodes indicate their pivot ranges within a triangle and ray list maintained on a per level basis. The highlighted node is marked for naive intersections and its contents are copied directly into the terminal node buffer at the end of the iteration*

The terminal nodes have their ids and data copied to a buffer and processed as described in section 2.2. We then re-index the pivot elements to account for the terminal nodes that have been removed from the list. We free memory used for the current level and proceed ahead to the new level created in a similar manner, till we have no more levels to process. We use $\alpha = 256$ and $\beta = 256$ for best performance.

2.2 Segmented Naive Ray-Triangle Intersections

The pseudocode for processing the terminal nodes is given in Algorithm 3. In order to fully utilize the resources of the GPU, we follow a buffered approach where each terminal node's triangle and ray ids are copied into a pre-allocated buffer that has a fixed size of η elements. When the buffer contents cross some fixed threshold $\lambda \leq \eta$(Line 7 in Algorithm 1), we launch a GPU kernel that computes naive intersections between all the rays and triangles in every stored node. Every ray is handled by one thread and blocks of *N* threads handle every segment(node). Triangles for each segment are loaded in batches of size *T* into shared memory(Line 5 in Algorithm 3). We then proceed to compute the minimum *mints* and corresponding *hitid* value of every ray in the buffer by sorting the buffer *mints* and *hitid* value by using the ray-id and then performing a segmented minimum reduction using the ray-id as key. We then launch a kernel that updates the *global mints* and *hitid* arrays appropriately. In our experiments we empirically set the value of *N* = 256 and *T* = 256.

Algorithm 2 Parallel Filter Algorithm

```
1:  procedure ParallelFilter(Aabb List, Element List, Element id List, Segment List, Pivot List)
2:      for each segment i ∈ (0, numsegments] in parallel do
3:          for each element j ∈ (0, numelements] in parallel do
4:              Split parent AABB into Left and Right
5:              Compute element intersection with AABB splits
6:              Compute status code and update keys
7:          end for
8:      end for
9:      parallel ReduceByKey(keys, keys, newkeys, newvals)
10:     for each key i in newkeys do
11:         Compute count of each status code
12:     end for
13:     for each i ∈ (0, numsegments] in parallel do
14:         Compute left, both, right and none values
15:         Update child pivot values
16:     end for
17: end procedure
```

Algorithm 3 Parallel Segmented Ray-Triangle Intersections

```
1:  procedure NaiveIntersection(TerminalNodeBuffer, Triangles List, Rays List)
2:      for each segment i ∈ (0, numsegments] in parallel do
3:          for each ray j ∈ (0, numrays] in parallel do
4:              Batch load triangle data into shared_memory[256]
5:              for each triangle k ∈ shared_memory do
6:                  Compute ray-triangle intersection
7:                  Update ray intersection parameters
8:              end for
9:          end for
10:     end for
11:     // Using buffer values
12:     parallel SortByKey(rayid, mints, hitids)
13:     parallel ReduceByKey(rayid, mints, hitids)
14: end procedure
```

3 Results

We have implemented our algorithm using CUDA on a machine having an Intel Core i7-920 CPU and a Nvidia GTX 580 GPU. All reported results are averages of values obtained during rendering from 12 different viewpoints.

3.1 Performance

Dynamic scenes change every frame. DACRT builds an implicit hierarchy and traces the hierarchy simultaneously. Table 1 compares the performance of our PDACRT method with the CPU DACRT work [Mora 2011]. The timings reported are for primary rays along with the implicit structure creation. Our algorithm performs better than CPU DACRT for all the scenes except the Conference scene. The early ray termination using cones makes the sequential CPU version fast for Conference, as the viewpoint is inside the bounding box. Our PDACRT does a lot of work lacking early termination. Inspite of this, we are 1.2 to 1.5 times faster than the CPU implementation on other scenes with viewpoints outside the bounding box. The Conference scene needs 344 ms without cone tracing on the CPU [Mora 2011]. The effect of parallel acceleration is thus much higher than is indicated in Table 1 as a result. PDACRT also performs much better compared to the best parallel CPU k-d tree method, which needs 654ms and 835ms respectively to build the k-d tree for Dragon and Buddha models on a 32 core machine [Choi

Scene	CPU DACRT	PDACRT
Bunny	105 ms	87 ms
Conference	99 ms	148 ms
Angel	n/a	143 ms
Dragon	n/a	155 ms
Buddha	238 ms	165 ms
Turbine	285 ms	223 ms

Table 1: *Comparison of ray casting time between CPU DACRT[Mora 2011] and Parallel DACRT(PDACRT). CPU DACRT results indicate ray casting time in a single core 3Ghz Core 2 machine. All scenes were rendered at 1024x1024 resolution*

Scene	PDACRT Shadow rays	PDACRT Specular Refl. rays	PDACRT AO rays
Bunny	67 ms	96 ms	149 ms
Conference	197 ms	222 ms	240 ms
Sponza	220 ms	246 ms	280 ms
Angel	102 ms	163 ms	182 ms
Dragon	128 ms	177 ms	192 ms
Buddha	150 ms	190 ms	204 ms
Turbine	213 ms	252 ms	287 ms

Table 2: *PDACRT performance results for shadow rays, specular reflection rays and ambient occlusion rays. All results were generated at* 1024×1024 *resolution. Shadow rays were generated with one point light source. Ambient occlusion rays were generated with 8 AO rays per primary ray intersection.*

et al. 2010]. The best GPU k-d tree construction method also needs 511ms and 645ms respectively for Dragon and Buddha [Wu et al. 2011]. PDACRT is 4-5 times faster to construct-and-trace these models. We also tested the performance of our algorithm for secondary rays with path tracing. Our path tracing performance with a 1024×1024 resolution and 7 diffuse bounces for the Conference and Sponza scenes are 5.9 MRays/sec and 4.8 MRays/sec respectively, compared to 5.5 and 4.6 MRays/sec reported by [Mora 2011]. Table 2 shows performance of PDACRT for shadow, specular reflection, and ambient occlusion rays. Shadow rays generated towards a point light source have little coherence. As can be seen from Table 2, PDACRT is a little slower on the secondary ray pass than the primary pass, with higher difference on small object models when the viewpoint is inside the bounding box due to the highly incoherent nature of the rays. The performance for ambient occlusion(AO) rays (Table 2) show that even large number of incoherent rays is handled well by PDACRT.

3.2 Memory Requirements

A key advantage of DACRT is its low memory footprint. The CPU implementation by [Mora 2011] follows a recursive depth first approach. Their memory requirements were for the two pivots (triangle and ray) and for the recursion stack only. Parallel DACRT implementation needs to store and maintain auxiliary information about all the nodes at any level during each iteration. More memory is needed as a result, but much lower than more traditional acceleration structures. Table 3 compares our peak memory usage with the GPU Kd-Tree [Wu et al. 2011] for primary rays at 1024×1024 resolution. All scenes were rendered with a buffer of size 24 MB except Bunny(8 MB buffer). In case of larger scenes or scenes with a large amount of rays, more memory would have to be allocated but still significantly lower than memory required by traditional acceleration structures.

Scene	PDACRT	GPU SAH KD-Tree
Bunny	47.66 MB	33.96 MB
Fairy	82.25 MB	80.33 MB
Exploding	82.68 MB	86.58 MB
Conference	85.82 MB	159.98 MB
Angel	82 MB	218.26 MB
Dragon	96.87 MB	417.33 MB
Buddha	107.89 MB	512.65 MB

Table 3: *Comparison of peak memory usage for various scenes between Parallel DACRT and GPU SAH KD-Tree [Wu et al. 2011] construction method. Parallel DACRT(PDACRT) values are for scenes rendered at 1024 x 1024 resolution. Parallel DACRT values include memory requirements for buffer storage, triangle and ray data(32MB) also. Values for GPU SAH KD-tree construction include memory for triangle data only and not ray data.*

3.3 Limitations

The level-wise processing of nodes results in duplication of ray ids which can be quite considerable at deeper levels, as a ray can intersect multiple nodes in a level. The duplication can be large for incoherent rays since rays can traverse deeply before they are culled. This behaviour was seen particularly for indoor scenes (like Conference) where the camera is inside the bounding box. All rays test positive for intersections at the upper levels of the hierarchy and have their ids duplicated. This duplication can lead to increased memory requirements when enough rays and triangles are not culled effectively. This behaviour is visible with relatively higher memory requirements for Fairy and Conference scenes (Table 3). Another issue is the lack of optimization strategies such as ordered traversal that is available when tracing individual rays through an acceleration structure. Parallel DACRT always processes rays in batches across all nodes in a level and hence ordered traversal is not possible.

4 Conclusions

In this paper, we have presented a fully parallel DACRT algorithm on the GPU which is the first implementation of its kind. In the future, we would like to optimize the algorithm further by employing recent advances in hardware and also perform better load balancing in a hybrid CPU-GPU setting.

References

ÁFRA, A. T. 2012. Eg 2012 - short papers. Eurographics Association, Cagliari, Sardinia, Italy, 97–100.

CHOI, B., KOMURAVELLI, R., LU, V., SUNG, H., BOCCHINO, R. L., ADVE, S. V., AND HART, J. C. 2010. Parallel sah k-d tree construction. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, HPG '10, 77–86.

MORA, B. 2011. Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph. 30*, 5 (Oct.), 117:1–117:12.

NABATA, K., IWASAKI, K., DOBASHI, Y., AND NISHITA, T. 2013. Efficient divide-and-conquer ray tracing using ray sampling. ACM, New York, NY, USA, HPG '13, 129–135.

WU, Z., ZHAO, F., AND LIU, X. 2011. Sah kd-tree construction on gpu. ACM, New York, NY, USA, HPG '11, 71–78.