

# A Hardware Pipeline for Accelerating Ray Traversal Algorithms on Streaming Processors

Michael Steffen  
Electrical and Computer Engineering  
Iowa State University  
steffma@iastate.edu

Joseph Zambreno  
Electrical and Computer Engineering  
Iowa State University  
zambreno@iastate.edu

**Abstract**—Ray Tracing is a graphics rendering method that uses rays to trace the path of light in a computer model. To accelerate the processing of rays, scenes are typically compiled into smaller spatial boxes using a tree structure and rays then traverse the tree structure to determine relevant spatial boxes. This allows computations involving rays and scene objects to be limited to only objects close to the ray and does not require processing all elements in the computer model.

We present a ray traversal pipeline designed to accelerate ray tracing traversal algorithms using a combination of currently used programmable graphics processors and a new fixed hardware pipeline. Our fixed hardware pipeline performs an initial traversal operation that quickly identifies a smaller sized, fixed granularity spatial bounding box from the original scene. This spatial box can then be traversed further to identify subsequently smaller spatial bounding boxes using any user-defined acceleration algorithm. We show that our pipeline allows for an expected level of user programmability, including development of custom data structures, and can support a wide range of processor architectures. The performance of our pipeline is evaluated for ray traversal and intersection stages using a *kd*-tree ray tracing algorithm and a custom simulator modeling a generic streaming processor architecture. Experimental results show that our pipeline reduces the number of executed instructions on a graphics processor for the traversal operation by 2.15X for visible rays. The memory bandwidth required for traversal is also reduced by a factor of 1.3X for visible rays.

## I. INTRODUCTION

Continuous advancements in real-time graphics hardware processing power and core count are having a dramatic effect on accelerating real-time ray tracing algorithms. Ray-tracing [1] [2] [3] is a rendering algorithm that uses rays to represent light in a 3D computer model. Typical implementations map each pixel to a ray and computes additional rays for shadows and reflections. Modern day graphics hardware supports ray tracing rendering algorithms as general purpose computations on graphics hardware (GPGPU). This results in a majority of the operations being implemented in software and only common shading operations, that are related to rasterization, being accelerated using fixed hardware. While GPGPU ray tracing algorithms have improved performance over CPUs, current implementations fall short of the GPUs theoretical performance by a factor of 1.5-2.5X due to memory bandwidth needs and inefficient work distribution on programmable processors [4].

We present a hardware pipeline designed to accelerate part

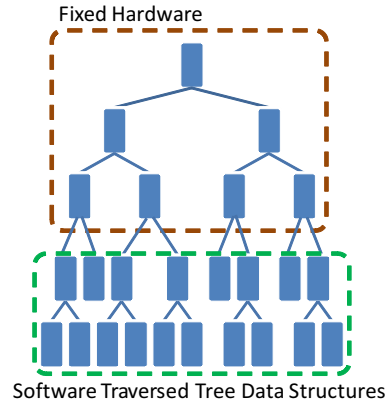


Fig. 1. Proposed new traversal for an arbitrary sized tree data structure. This results in smaller tree data structures that processors have to traverse.

of the ray tracing algorithm using custom fixed hardware that reduces both the computational load on programmable processors as well as the overall memory bandwidth. Our pipeline is designed to accelerate the traversal of custom data structures used for storing the scene geometry. By targeting this stage of the ray tracing process, we can insert fixed hardware components with minimal impact on the programmability that users expect. Instead of having the software traversal algorithm start with the entire scene data structure (equivalent to starting at the root node of the acceleration tree structure), our pipeline allows for the scene to be broken up into smaller groups which are the starting points for software traversal. This results in improved performance by reducing the workload required by the graphics processors which are the bottleneck for rendering time.

Our proposed two-stage traversal process is illustrated in Figure 1. The first stage is implemented in hardware and can quickly traverse to a coarse-grain bounding box located in the original scene. The second stage is the user-defined algorithm implemented on the graphics processors that takes as input the coarse-grain bounding box and traverses it to completeness. The hardware implemented traversal stage is accessible through a new instruction in the Instruction Set Architecture (ISA). The results from our pipeline is a memory reference to the coarse-grain bounding box that is used as

the root node for user-defined traversal algorithms. If no valid intersection is found during the user-defined traversal algorithm, the new instruction can be called again and the next coarse-grain scene spatial box is identified.

Our pipeline offers the following advantages:

- **Maintains user programmability.** Ray-tracing algorithms require numerous programming stages in the rendering algorithm for both specialized graphics effects and scene dependent acceleration techniques. Our fixed hardware pipeline, while designed to accelerate traversal operations, still allows for this stage to be programmable. By performing the first part of the ray traversal in fixed hardware, the remaining operations that finish the traversal operations are implemented in user-defined code. Leaving the remaining parts of the the traversal operation programmable allows for the programming of custom acceleration methods and data structures.
- **Increases ray tracing performance.** The rendering time for ray tracing algorithms is greatly dependent on the performance of the acceleration structure. Implementing this stage in fixed hardware accelerates the traversal process and decreases the programmable processor workload.
- **Diverse implementation scope.** Our fixed hardware pipeline is a standalone processing element that connects to graphics processors through the ISA. This allows for easy portability between different processors, allowing for compatibility for multiple processor architectures and programming models.

A custom performance simulator was created for our fixed-hardware pipeline data structure and graphics processors. Our simulator implementation uses a streaming processor architecture and a *kd*-tree as the software acceleration algorithm. Our experimental results using this simulator show a reduction in the number of software down traversal operations of up to 32X. The overall instructions executed on the graphics processor for traversal operations is reduced by an average of 2.15X for visible rays.

The remainder of this paper is organized as follows: Section II discusses previous work in data structures and graphics hardware. Section III describes our data structure and traversal algorithm. Section IV outlines our pipeline architecture and Section VI shows the results of our simulator using eight benchmark scenes. Finally Section VII concludes the paper with a description of planned future work.

## II. PREVIOUS WORK

To accelerate rendering time, a variety of rendering methods [5] [6] [7] use accelerated data structures and parallelism offered by hardware. In the past, data structures and hardware have been developed separately; however in the near future it is expected that graphics hardware will migrate from *z*-buffer rendering towards directly incorporating scene data structures for ray tracing [8].

### A. Accelerated Data Structures

The most widely used acceleration data structures are implemented using Hierarchical Space Subdivision Schemes (HS3) like *kd*-tree [9] and Bounding Volume Hierarchies (BVHs) [10] that use tree data structures in memory. These data structures are popular because they allow the data structure to adapt to the scene’s spatial layout of objects, allowing for more uniform distributions of scene objects in the data structure.

Current research advancements in this field have focused on improving performance and customizing these algorithms for specific hardware implementations. Algorithm advancements have focused on different creation methods for tree data structures that reduce the number of operations required for ray traversal [11] [12]. GPU implementations are a common research topic because of their availability and large multi-core processor count. Implementations such as [13][14][15] focus on customizing current acceleration structures to match the GPU processor architectures. Additional hardware acceleration for data structures use vector operations for a given processor. Quad-BVH [16] utilizes a processor’s vector-width to convert binary trees into quad-based trees, reducing the size of the tree and the number of traversal steps.

### B. Graphics Hardware

Graphics hardware research has focused on large programmable multi-core architectures. Intel’s Larrabee [17] is one such architecture, composed of several in-order *x86* SIMD cores that can be programmed to support any graphics pipeline in software. Acceleration comes from running large amounts of graphic computations in parallel and running multiple threads on each processor to reduce the latency for memory operations. NVIDIA’s Compute Unified Device Architecture (CUDA) [18] and Copernicus [19] also offer large numbers of cores and can hide memory latency through using large numbers of threads with no switching penalty. Ray tracing implementations on these architectures is accomplished through software kernels that then run on the processors. Other multi-core architectures for ray tracing include SaarCore [20] [21], RPU [22] [23] and TRaX [24].

Common methods for hardware acceleration of ray tracing data structures has been through programmable processor architectures containing vector operations or vectorized processors. The current trend for graphics hardware focuses on increasing programmable processor counts, allowing for acceleration through increased parallel computations only. This paper presents a tighter integration for accelerating data structure operations using fixed hardware while conforming to current programming trends.

## III. PIPELINE TRAVERSAL ALGORITHM

Our pipeline traversal algorithm, called Group Uniform Grid (GrUG), is based on the algorithm presented in [25]. In this paper we explore the hardware design space of the GrUG algorithm. As a result of this, a new algorithm better suited for hardware implementation was created. This section presents

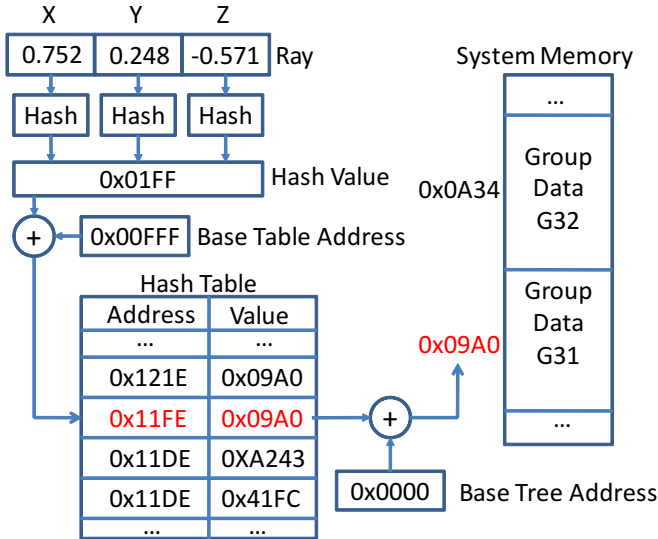


Fig. 2. Hash function starting with X,Y,Z coordinates and outputting the memory address of a GrUG grouping that can be passed to a software traversal algorithm.

a new algorithm, that still uses the concept of a hash-table presented in previous work, but redesigns the supporting operations and data structures for hardware implementation. This new algorithm allows for easier implementation in multi-core processor architectures and has a smaller hardware footprint.

### A. Overview of GrUG

GrUG uses two spatial separation methods to divide a scene into multiple non-overlapping smaller scenes. The first spatial separation method is a uniform grid. Properties of a uniform grid allow rays to identify which cell of the uniform grid contains the ray with no memory reads and few computations. The second spatial separation method accounts for the uniform grids' inability to adapt to scene geometry by grouping cells in the uniform grid. This forms cubical groupings referred to as GrUG groups. By grouping uniform grid cells, the resolution of a GrUG group is based on the uniform grid size. This often prevents GrUG from being an effective standalone traversal method. GrUG groups are created from uniform grid cells by combining similar cells, allowing for the GrUG groups to adapt to scene geometry. GrUG groups are the small scenes that become the root nodes for the user-defined software traversal algorithms. A GrUG grouping can consist of only a single uniform grid cell, the entire scene, or anything in between.

To traverse GrUG, rays must be mapped to a uniform grid cell and the uniform grid cell must be mapped to a GrUG group. A hash table is used for performing both mappings, where rays are the input value and the values stored in the hash table are the memory values to the GrUG group (the root node for user-defined traversal algorithm). Figure 2 shows the steps used for our hash table. The hash function takes as input the ray location and outputs the uniform grid cell ID using a custom hash function. Cell IDs are arranged such that the three

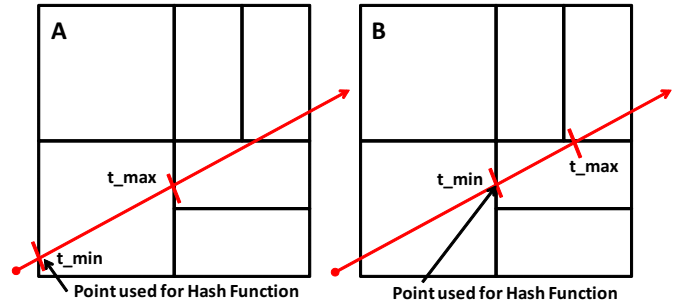


Fig. 3. Ray projection from original GrUG grouping in A to next GrUG grouping in B. To compute the next point along the ray for the hash function, the ray is projected by the  $t_{min}$  value.

indexes of a cell along the three axes can be concatenated to form a cell ID. This allows for parallel processing of the values used for ray location. For inputted rays to output the cell ID in which the ray is positioned, the hash function requires that all collisions result in the same uniform grid cell. Figure 5 shows the hash function algorithm. The results from the hash function can be used as a memory address either as a pointer or a memory offset value. The memory address stored in the hash table is the resulting GrUG grouping where the ray is located.

Having the root node and ray is not enough for performing additional traversal operations or ray-triangle intersection testing. The values of  $t_{min}$  and  $t_{max}$  must be known for proper results. The  $t_{min}$  and  $t_{max}$  values are used to represent a valid part of the ray inside of a bounding box (see Figure 3). The  $t_{min}$  value is known at the start of each traversal from having  $t_{min}$  start at 0.0 for new rays or setting its value to be  $t_{max}$  from previous GrUG traversal operations. The  $t_{max}$  value

must be computed for each ray in a GrUG grouping. To compute  $t_{max}$ , the bounding box for each GrUG grouping must be known and is stored in the GrUG group node.

If a ray does not find an intersection in the current GrUG grouping, the ray must continue through the scene until it exits the entire scene or passes into another GrUG grouping. GrUG traversal can be used again on a ray, but the location of the ray must be updated, otherwise it would produce the same results. To update the locations, rays are projected forward using the value of  $t_{min}$  as shown in Figure 3. To keep the original value of the ray, a copy of the ray is made, projected, and inputted into the hash function. For newly created rays,  $t_{min}$  will be zero, producing no projection. If a ray requires additional GrUG traversal, the  $t_{min}$  value should be set to the  $t_{max}$  value from the previous traversal operation, allowing the ray to be projected out of its previous GrUG grouping.

### B. Data Structure Creation

Creating the GrUG and user-defined tree data structure requires setting up two memory spaces, the hash table used by GrUG and a user-defined tree data structure. Since the hash table memory contains pointers to the user-defined tree data

structure, the tree data structure is setup first and then the hash table memory is populated with pointers to the appropriate tree nodes. With the tree data structure starting at GrUG groupings, we define the GrUG groupings first. A traditional  $kd$ -tree creation algorithm is used to create GrUG groupings from the  $kd$ -tree leaf nodes with two exceptions. One, because GrUG operates on a uniform grid structure, splitting locations of the  $kd$ -tree structure must align with these bounds until a leaf node is identified. Two, only the leaf nodes are preserved in memory, as GrUG will map to these leaf nodes and not the  $kd$ -tree nodes. By forcing the  $kd$ -tree structure to align with GrUGs uniform grid cells, GrUG groupings are composed of unique uniform grid cells that neither overlap or leave gaps in the scene. Once a GrUG grouping has been identified, the bounding box can be stored in memory and the hash table can be populated for all uniform cells contained in this GrUG grouping. GrUG groupings can also continue being subdivided using a custom acceleration structure creation algorithm with no restrictions.

#### IV. PIPELINE ARCHITECTURE

Our pipeline stages are responsible for performing the first of two data structure traversal algorithms on rays. The ray projection and hash function are implemented in fixed hardware, and on-chip cache for storing the hash table. Programmable processors are then used to read from the GrUG group bounding box. To allow the GrUG pipeline to be applicable to a wide range of processor architectures, our implementation is a stand-alone processing block inside of a processor that is accessible through the ISA.

##### A. Fixed Hardware

The hardware pipeline stages for GrUG traversal are shown in Figure 4. Memory-addressed registers are used for configuring initial parameters of the pipeline for the size of the grid resolution and are set at the start of the application. The first stage of the pipeline is ray projection. Rays requiring additional traversal operations (rays that did not find an intersection in an earlier GrUG traversal) need to be projected out of the original GrUG grouping. Rays are projected by the  $t_{min}$  value using 3 floating point multiply accumulators, one for each axis. This requires that when rays are finished with the user-defined traversal algorithm,  $t_{min}$  must be set to the  $t_{max}$  value, resulting in the ray being projected out of the current scene grouping. Next, the ray undergoes GrUG traversal using the hardware hash function. The result of the hash function is used to read the entry in the hash table that is stored in an on-chip cache. Cache misses stall the pipeline and fetch the value from global memory. The programmable processors are then used to read the bounding box of the GrUG groups that are stored in the tree data structure located in device memory and the  $t_{max}$  value is computed (see Algorithm 1).

The pipeline is designed for a large throughput of rays and allows for a ray to be outputted every clock cycle. The number of cycles corresponding to the individual stages are shown in Table I. To allow for an even higher throughput of

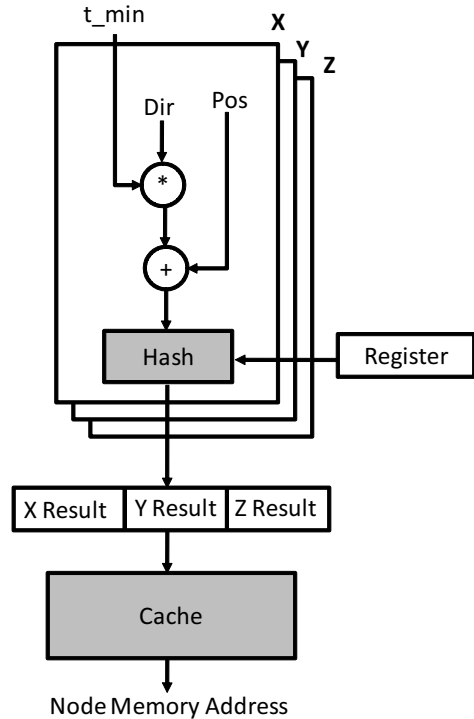


Fig. 4. Integration of the GrUG pipeline into a multi-core graphics processor and the fixed hardware stages for the GrUG pipeline.

Pipeline Stage	Pipeline Depth	Throughput
Ray Projection	2	1
Hash Function	5	1

TABLE I  
GRUG PERFORMANCE PARAMETERS FOR EACH FIXED HARDWARE PIPELINE STAGE.

rays, each pipeline stage can be vectorized, allowing multiple rays to be processed in parallel for each pipeline stage. This configuration is ideal for wide streaming processors requiring execution of different threads every clock cycle resulting in repeating instruction calls to the pipeline.

##### B. Hash Function

The hash function's responsibility is to determine in which GrUG uniform grid cell a ray belongs to. The resulting uniform grid cell ID is converted to a memory address that the GrUG software algorithm uses to determine the location of the root node for the user-defined traversal algorithm. The hash function hardware shown in Figure 5 takes as input one single precision floating point value representing the ray location for one scene axis. To support all three axes, three hash function pipelines are used in parallel and the results are concatenated to form one cell ID. The output for each hash function pipeline is a 9 bit value, resulting in a maximum grid size of 512 x 512 x 512. Smaller grid sizes are supported by truncating the least significant bits for each pipeline output. The hash function hardware, composed of integer subtraction and shift registers, is compatible with all floating point values from

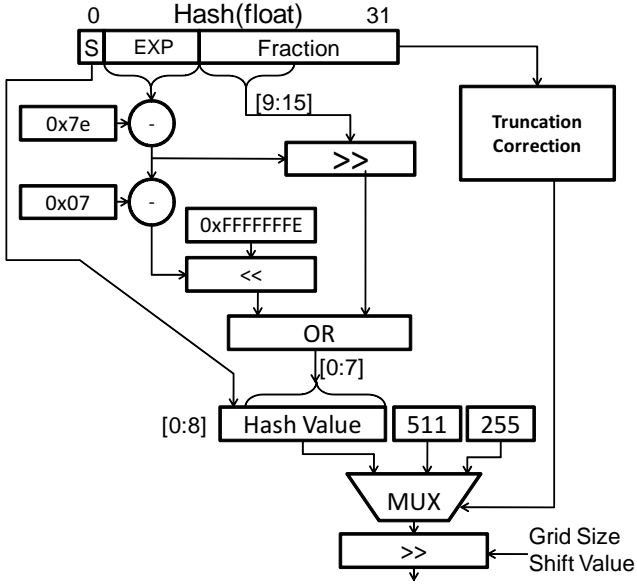


Fig. 5. Architecture of GrUG hash function for one axis using a 512 grid.

Shader Cores per Processor	8
Processor Cores	30
SIMD Warp Size	16
Memory Modules	8
Bandwidth per Memory Module	8 Bytes/Cycle
L1 Memory Caching	None
GrUG Cache	64KB 2-way set association

TABLE II  
CONFIGURATIONS USED FOR SIMULATION OF GRUG AND  
RADIUS-CUDA.

1.0 to  $-1.0$ . Values outside of this range are not supported, requiring all scenes to be scaled to fit into a 1.0 to  $-1.0$  sized box. Our hardware pipeline operates correctly for all floating point values between these limits except for two exceptions, values ranging from 0.5 to 0.503906 and  $-0.5$  to  $-0.503906$ . In these cases the fraction bits are zero and the exponential value results in hash value being equal to zero. At the same time, values close to  $\pm 0.25$  also result in a hash function of zero for similar reasons. To correctly identify these ranges, truncation correction is used to output corrected values for the  $\pm 0.5$  range allowing for each uniform grid cell to have a unique hash value.

## V. IMPLEMENTATION

Our presented hardware pipeline is designed for only accelerating traversal of acceleration structures, therefore only the traversal of acceleration structure and the ray intersection stages are simulated. Shading on graphics hardware is well established due to its usage in the conventional rasterization pipeline [26], and consequently was not simulated.

### A. Simulator

To evaluate the performance of our proposed pipeline hardware, the GPGPU-Sim simulator [27] was modified to

support our traversal pipeline. The simulator was configured with the parameters shown in Table II. These configurations resemble a GPU streaming processor able to run NVIDIA PTX [28] assembly code. PTX assembly files were generated using NVIDIA NVCC [18] compiler without using our fixed hardware pipeline since the compiler does not support our pipeline instruction. PTX assembly code was then modified to support our pipeline instruction.

---

### Algorithm 1 Ray Tracing with GrUG Pipeline

---

**Require:** *Thread ID*

**Ensure:** *pixel color*

*ray*  $\leftarrow$  createRay(*Thread ID*)

**if not** intersectSceneBox(*ray*) **then**

**return**

**end if**

*t<sub>min</sub>*  $\leftarrow$  max(intersectBoxEnter(*ray*), 0)

*t<sub>absMax</sub>*  $\leftarrow$  intersectBoxExit(*ray*)

**while** *t<sub>min</sub>* < *t<sub>absMax</sub>* **do**

*node*  $\leftarrow$  GrUGHardwareTrav(*ray*, *t<sub>min</sub>*)

*t<sub>max</sub>*  $\leftarrow$  computeMaxT(*node*.BBox, *ray*)

*hit*  $\leftarrow$  userDefTrav(*ray*, *node*, *t<sub>min</sub>*, *t<sub>max</sub>*)

**if not** *hit* **then**

*t<sub>min</sub>*  $\leftarrow$  *t<sub>max</sub>*

**else**

**break**

**end if**

**end while**

*pixel color*  $\leftarrow$  shade(*ray*)

---

### B. Kernel Code

Four software functions were written for our ray tracing application that was used for simulation: Ray Generation, post GrUG traversal operations, *kd*-tree algorithm representing the user-defined traversal algorithm and ray-triangle intersection. The post GrUG traversal code performs the memory operations required to read the selected GrUG grouping bounding box and compute the ray's *t<sub>max</sub>* value. The final step of this function is to call the user-defined traversal algorithm and ray-triangle intersection algorithm. Algorithm 1 shows the algorithm used for our application. The user-defined traversal algorithm for simulation is a *kd*-tree from the Radius-CUDA program [6]. The ray-triangle intersection algorithm is Wald's [29] algorithm.

### C. Benchmark Scenes

A total of eight scenes were tested at a resolution of 512 x 512 with varying polygon counts and grid sizes. Table III lists the benchmarks with scene data. All scenes cover a wide range of scene possibilities from lower polygon count, uniform polygon layout, teapot in stadium effects and large polygon count.

## VI. RESULTS

For comparison purposes, the Radius-CUDA program (a ray tracing application using NVIDIA CUDA) was modified to only perform ray traversal operations and use the same ray

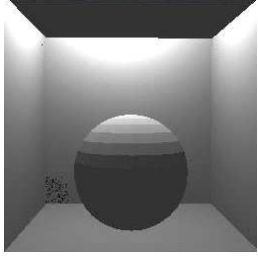
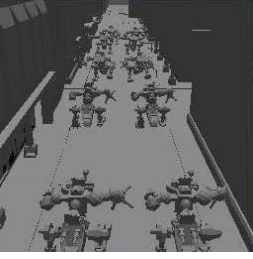
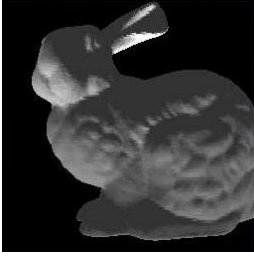





				
Benchmark	UlumBox	Robots	Stanford Bunny	Kitchen
Triangles	492	69,296	69,451	110,540
kd-tree Depth	17	33	27	31
GrUG Tree Depth	4	17	8	13
				
Benchmark	Fairyforest	Atrium	Conference	Happy Buddha
Triangle Count	172,561	559,992	987,522	1,087,716
kd-tree Depth	36	37	35	34
GrUG Tree Depth	21	19	20	14

TABLE III  
BENCHMARK SCENES WITH TRIANGLE COUNT AND TREE DATA STRUCTURE PARAMETERS.

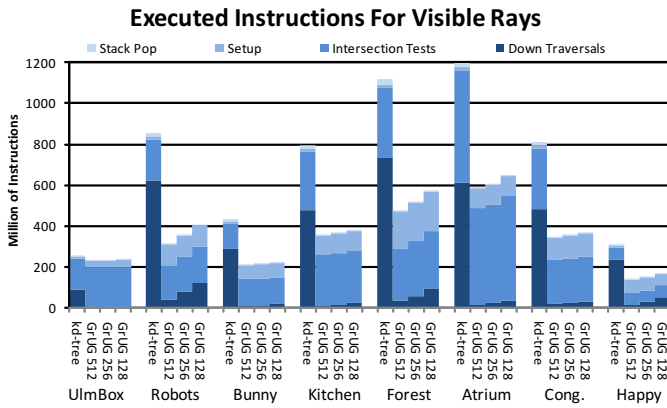


Fig. 6. Total number of instructions executed for traversal of visible rays. kd-tree results are compared against GrUG using grid sizes of 512, 256 and 128.

intersection algorithm [6]. Performance results for Radius-CUDA were measured using our simulator. It is important to note that our CUDA kernels are not optimized for performance; therefore, all performance results will be a comparison between our simulation results and our modified Radius-CUDA program.

#### A. Performance

The overall performance results are shown in Figure 6. By using our hardware pipeline we reduce the number of tree traversal steps by an average of 32.5X for visible rays.

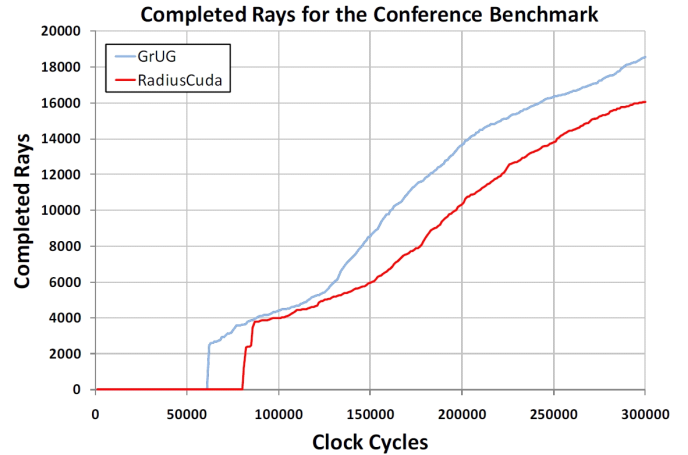


Fig. 7. Number of completed rays for the first 300,000 clock cycles of the Conference benchmark scene.

The overall speedup for traversal using GrUG is an average of 1.6X for visible rays, with a maximum of 2.74X for the Robots benchmark. This reduction in executed instructions is due to shallower tree data structures as shown in Table III. While the tree traversal executed instructions has drastically decreased, the GrUG software traversal kernel becomes the most time consuming part of the traversal algorithms. The GrUG software traversal kernel requires two back-to-back device memory reads. The first operation retrieves the root



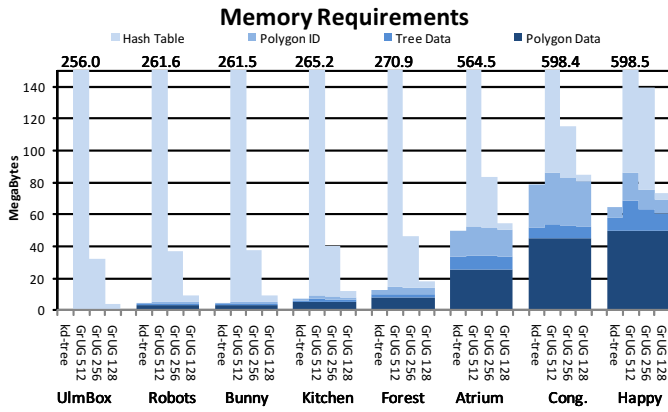


Fig. 8. Memory requirements for kd-tree and GrUG.

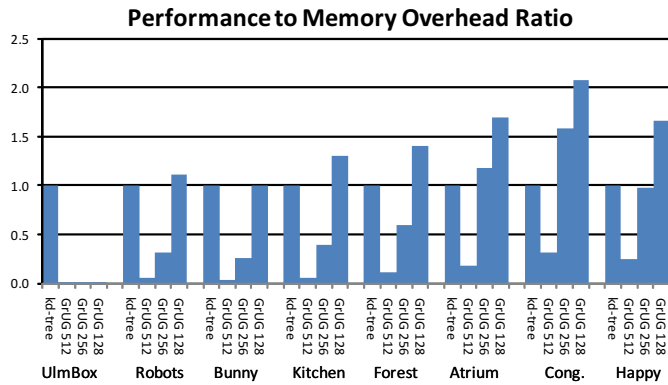


Fig. 9. Ratio of the performance increase divided by the memory overhead for different benchmarks and grid sizes. Performance increase and memory overhead are based on the kd-tree algorithm resulting in a value of 1.0 for the kd-tree.

node tree data structure and the second operation reads the root node boundary values for computing  $t_{max}$ . In addition, the GrUG software kernel is called more times than there are down traversal operations, due to smaller tree data structures and that many of the GrUG groups do not require a tree data structure. Using smaller grid sizes for GrUG results in a higher number of software tree traversal steps and an increase in the number of executed instructions. While adding software tree traversal steps increases the run time, the performance for a grid size of 128 is still improved over software implementations by 1.9X compared to 2.15X for a grid size of 512.

In addition to comparing the number of executed instructions, we simulated the first 300,000 cycles for the Conference benchmark scene using a resolution of 128. Figure 7 shows the number of finished rays during this time period. The GrUG hardware pipeline was able to compute 2,500 more rays than the Radius-CUDA implementations.

### B. Memory

Figure 8 shows the total system memory needed to store the different data structures used for combining GrUG and kd-tree. The use of a hash table in GrUG results in a significant overhead in memory requirements to store the entire hash table

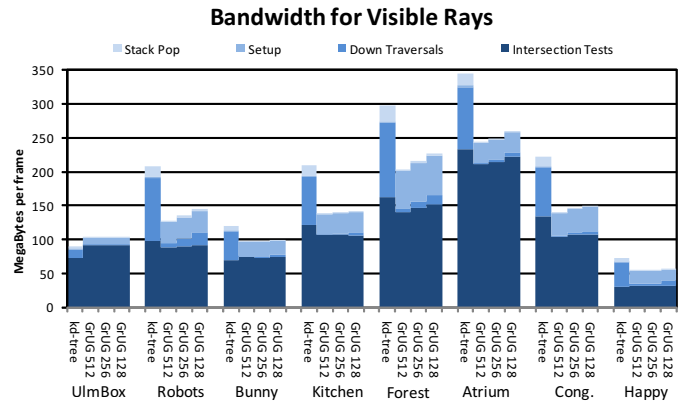


Fig. 10. Memory bandwidth per frame required to render visible rays without caching.

in memory. The required memory is based on the GrUG grid size, 4 bytes per grid cell. A fixed grid size of 512 will use 512MB to store the hash table. Four bytes per grid cell is a conservative usage resulting in a maximum of 4,294,967,296 supported GrUG groups. Scenes requiring 65,536 or fewer GrUG groups can use 2 bytes per grid cell, resulting in only a 256MB hash table. The use of smaller grid sizes reduces the memory requirements for the hash table down to 4MB for the 128 size grid. On average, using a grid size of 128 requires only 1.5 times the memory of a kd-tree, compared to a grid size of 512 that requires 27.6 times the memory of a kd-tree. Figure 9 shows the ratio of the performance increase divided by the memory overhead for different grid sizes. Having larger grid sizes offer better overall performance, but the memory overhead grows faster than the performance benefit. Using smaller grid sizes offer a nice balance of increasing performance to the increased memory requirements.

In addition to the hash-table, the kd-tree structure and bounding dimensions of all threshold nodes must be stored in system memory. While GrUG requires a smaller tree data structure, the memory requirements are similar to the full kd-tree data structure. An additional 24 bytes are needed for storing the bounding dimensions per GrUG grouping, equivalent to 3 nodes. This results in similar memory requirements for storing the tree data structures as a full kd-tree.

### C. Bandwidth

While GrUG requires more memory storage than standalone tree data structures, the average memory bandwidth per frame is smaller. Figure 10 show the memory bandwidth for each benchmark without caching. Since we are reducing the number of down tree traversal steps, the amount of device memory transactions required is significantly less. The use of GrUG results in a majority of the bandwidth required for rendering a frame being used for post GrUG software traversal instead of user-defined traversal due to the number of memory reads from the boundary box and root node for every ray outputted from the hardware pipeline. However the sum of the memory bandwidth of GrUG and down tree traversal is smaller than the

down traversals required by a full software implementation.

## VII. CONCLUSIONS AND FUTURE WORK

The demand for improved visual quality and additional interaction for computer graphics has resulted in research in new graphics hardware architectures. While no direct substitute has been defined for rasterization, new rendering algorithms will most likely require further programmability in implementation and require processing of large amounts of independent rays. To match these needs, graphics hardware is shifting from fixed hardware pipelines to massive multi-core architectures. In this work we proposed adding a small fixed-hardware pipeline designed to accelerate part of the ray tracing algorithm. Our pipeline is designed to work with multi-core graphics hardware and offload part of the acceleration traversal computations. We have shown that our fixed hardware pipeline still allows for a diverse implementation scope of processor architectures and still offers the same user programmability that multi-core implementations support. With our pipeline relying on existing processor architectures, advancements in both multi-core architectures and algorithms will also benefit the overall runtime performance using our fixed pipeline.

We have evaluated our fixed hardware pipeline using a simulator modeling a streaming processor architecture. Our results show that we can improve the traversal operations by 2.15X for visible rays. While the best performance for GrUG requires significant amounts of memory, using a grid size of 128 only requires on average 1.5X of memory compared to kd-tree and still offers significant performance improvements.

While this paper focused on only ray traversal and intersection test, future work is to implement a full graphics processor simulator to allow for additional rendering kernels, such as ray generation and shading. With a full rendering pipeline implementation, complete rendering time and frames-per-second can be determined along with allowing us to study different caching techniques to further improve traversal operations.

## ACKNOWLEDGMENT

This work has been supported in part by a National Science Foundation (NSF) Graduate Research Fellowship.

## REFERENCES

- [1] P. Dutre, P. Bekaert, and K. Bala, *Advanced Global Illumination*. Natick, MA, USA: AK Peters, Ltd., 2002.
- [2] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.
- [3] A. S. Glassner, Ed., *An introduction to ray tracing*. London, UK: Academic Press Ltd., 1989.
- [4] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of High-Performance Graphics*, 2009, pp. 145–149.
- [5] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.
- [6] Benjamin Segovia, "Radius-CUDA," 2008, <http://www710.univ-lyon1.fr/~bsegovia/demos/radius-cuda.zip>.
- [7] D. Luebke and S. Parker, "Interactive ray tracing with CUDA," NVIDIA Technical Presentation, SIGGRAPH, 2008.
- [8] W. R. Mark and D. Fussell, "Real-time rendering systems in 2010," in *ACM SIGGRAPH Courses*, 2005, p. 19.
- [9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [10] P. Shirley and R. K. Morley, *Realistic Ray Tracing*. Natick, MA, USA: AK Peters, Ltd., 2003.
- [11] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of High-Performance Graphics*, 2009, pp. 7–13.
- [12] S. Popov, I. Georgiev, R. Dimov, and P. Slusallek, "Object partitioning considered harmful: space subdivision for BVHs," in *Proceedings of High-Performance Graphics*, 2009, pp. 15–22.
- [13] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM SIGGRAPH Courses*, 2005, p. 268.
- [14] T. Foley and J. Sugerma, "KD-Tree acceleration structures for a GPU raytracer," in *Proceedings of Graphics hardware*, 2005, pp. 15–22.
- [15] D. R. Horn, J. Sugerma, M. Houston, and P. Hanrahan, "Interactive kd tree GPU raytracing," in *Proceedings of Interactive 3D graphics and games*, 2007, pp. 167–174.
- [16] H. Dammert, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays," *Computer Graphics Forum*, vol. 27, no. 4, pp. 1225–1233, 2008.
- [17] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerma, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH*, 2008, pp. 1–15.
- [18] NVIDIA, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0," 2007.
- [19] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. Mark, "Toward a multicore architecture for real-time ray-tracing," in *Proceedings of IEEE/ACM Symposium on Microarchitecture*, 2008, pp. 176–187.
- [20] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR: a hardware architecture for ray tracing," in *Proceedings of Graphics Hardware*, 2002, pp. 27–36.
- [21] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Real-time ray tracing of dynamic scenes on an FPGA chip," in *Proceedings of Graphics Hardware*, 2004, pp. 95–106.
- [22] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 434–444, 2005.
- [23] S. Woop, E. Brunvand, and P. Slusallek, "Estimating performance of a ray-tracing ASIC design," *Symposium on Interactive Ray Tracing*, pp. 7–14, 2006.
- [24] J. Spjut, D. Kopta, S. Kellis, S. Boulos, and E. Brunvand, "Trax: A multi-threaded architecture for real-time ray tracing," in *Proceedings of Application Specific Processors*, 2008, pp. 108–114.
- [25] M. Steffen and J. Zambreno, "Design and evaluation of a hardware accelerated ray tracing data structure," in *Proceedings of Theory and Practice of Computer Graphics*, 2009, pp. 101–108.
- [26] J. Montrym and H. Moreton, "The GeForce 6800," in *Proceedings of IEEE/ACM Symposium on Microarchitecture*, 2005, pp. 41–51.
- [27] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of Performance Analysis of Systems and Software*, 2009, pp. 163–174.
- [28] NVIDIA, "NVIDIA Compute PTX: Parallel Thread Execution ISA 1.1," 2007.
- [29] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Computer Graphics Group, Saarland University, 2004.