LITERATURE REVIEW: CUDA Accelerated Raytracing

Sivakumaran Gowthaman
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
sivakumarangowthaman@cmail.carleton.ca

December 20, 2020

Abstract

In CUDA accelerated raytracing implementation, when threads are called recursively to carry ray-surface intersection and shading functions, stack overflow occurs due to the limited shared memory of the GPU. This results in low quality rendered images. This paper addresses this issue by converting the recursive functions into iterative functions with explicitly defined stack depth. Redundant ray-surface intersection restricts the raytracing implementation from achieving real time rendering. This paper handles this issue by deploying Binary Volume Hierarchy (BVH) acceleration structure.

1 Introduction

Even though the modern Graphics Processing Units (GPUs) facilitates general programming computations in parallel manner, the paramount requirement which resulted in devising these GPUs was to render imageries for computer display. Rendering is the process of synthesizing a 2D image for a computer display from a 3D scene. It is the most popular form of human-computer interface and will undoubtedly continue to be so in the foreseeable future. The two prominent rendering methods are rasterization and raytracing. The first ever application that ran on GPUs was rasterization.

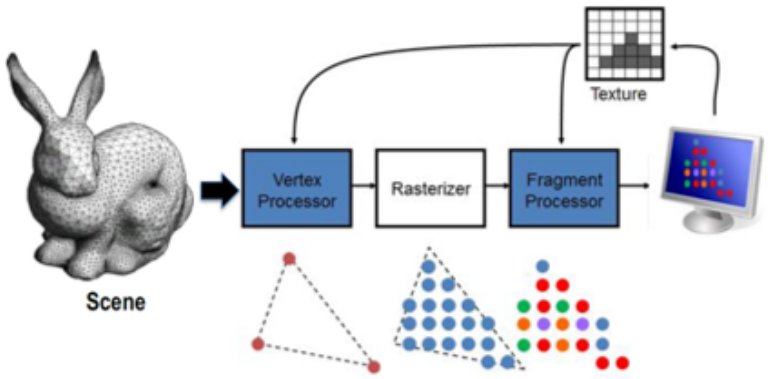

Figure 1: Rasterization Pipeline [11]

1.1 Rasterization

Rasterization is the popularly available general purpose rendering technique so far [3].

In rasterization, primitive's vertices are identified first and their respective encapsulated area is rasterized. Then using the fragment shader, the colour of the pixels are determined. When multiple primitives are found within the scene, the primitive closer to the virtual screen is rendered with high priority over the primitives further away from the virtual screen using the z-depth buffer

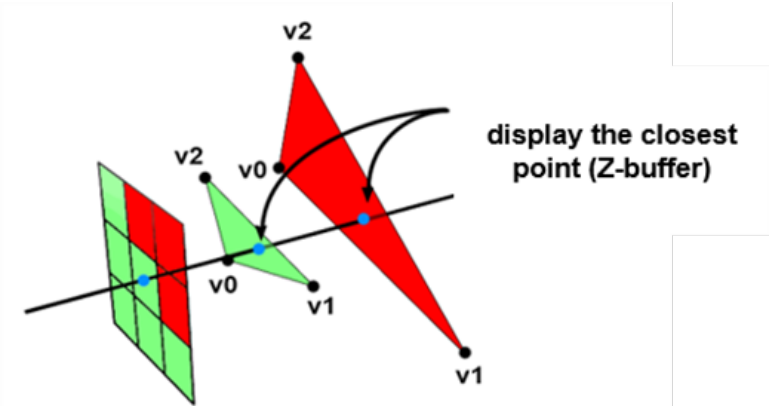

Figure 2: Z-buffer Calculation

Rasterization is the dominant rendering technique so far [3]. Most optical rendering effects, such as shadow, reflection, and refraction, can, however, only be produced by simultaneously considering multiple primitives [41]. By providing a natural way to synthesize visual effects and indirect lighting, ray tracing is the alternative superior mechanism for photo-realistic rendering [17]. In high-end graphics systems such as movie special effects, ray tracing-based rendering engines have been commonly used [9][12].

1.2 Raytracing

Raytracing is based on an emulation of the fundamental theory of vision. An object is observed by an observer (e.g., a human being or a camera) when it reflects the rays emitted from a light source to our eye. This procedure is done in reverse direction in raytracing so that the rays that do not reach our eyes do not need to be considered. As highlighted in Figure 3, the principle is to mimic the method of emitting rays from the eye toward pixels on the display screen. When a ray intersects an object in the 3D scene, the coloring of the corresponding pixel is determined by the illumination and the material at the specified intersection point.

Rasterization is less computation intensive and can attain interactive rendering rates when compared to ray tracing. On the other hand, ray tracing is superior in photo realistic quality but high computation intensive. This difference in photo realistic quality of images rendered using rasterization engine and raytracing engine developed for this project can be observed in Figure 4.

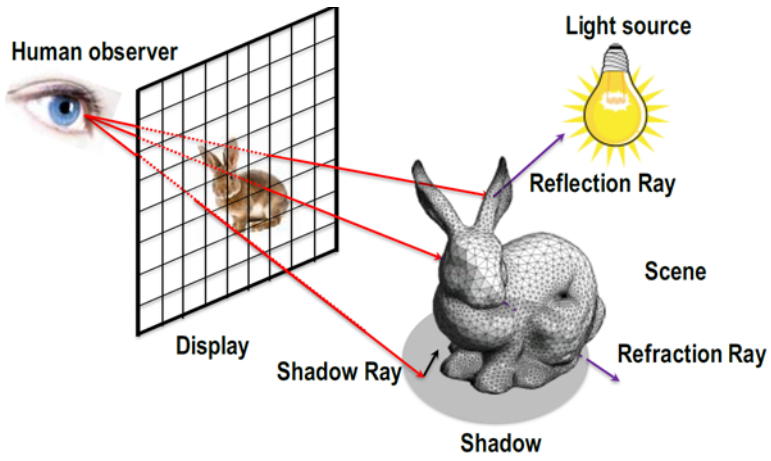

Figure 3: Raytracing [11]

The detailed clay material definition of the raytraced image showcases the superior photorealistic quality than the general plastic like rasterized image. Therefore achieving real time rendering using raytracing is the next big thing in human-computer interactive visualization. In this literature review acceleration techniques that are implemented to achieve real time interactive raytracing rendering using CUDA enabled GPUs are compared. The Three previously published techniques that incorporate ray tracing with spatial data structures on a GPU are contrasted in this literature review. There are uniform grids, kd-trees, and bounding volume hierarchies. While the algorithms were successfully mapped to a GPU, their performance on a current programmable GPU architecture has not been closely compared

2 Literature Review

2.1 Raytracing

For over four decades, research has been performed on ray tracing. Appel [5] first suggested the ray tracing algorithm. Since it just shoots rays from the eye and then scans the intersection, the earliest iteration was designated as the ray casting algorithm. Clearly, it only permitted the display of an approximate picture of the scene. To solve the visibility problem, this technique is widely applied. A ray-tracing algorithm was introduced by Whitted [53] to implement more complex illumination effects. A primary ray may emit so-called secondary rays, including reflection, refraction, and shadow rays, when a primitive is hit by that primary ray in the scene. In a recursive way, those rays can then be processed.

For real-time applications, the Whitted algorithm [53] is the prominent ray tracing algorithm. There are many implementations of GPU ray traversal in earlier works. Purcell et al. [43] explained how a GPU ray tracing pipeline can be applied over a paradigm of stream programming. As an acceleration framework, they used a Uniform Grid, expanding their ray tracer into stream programming.

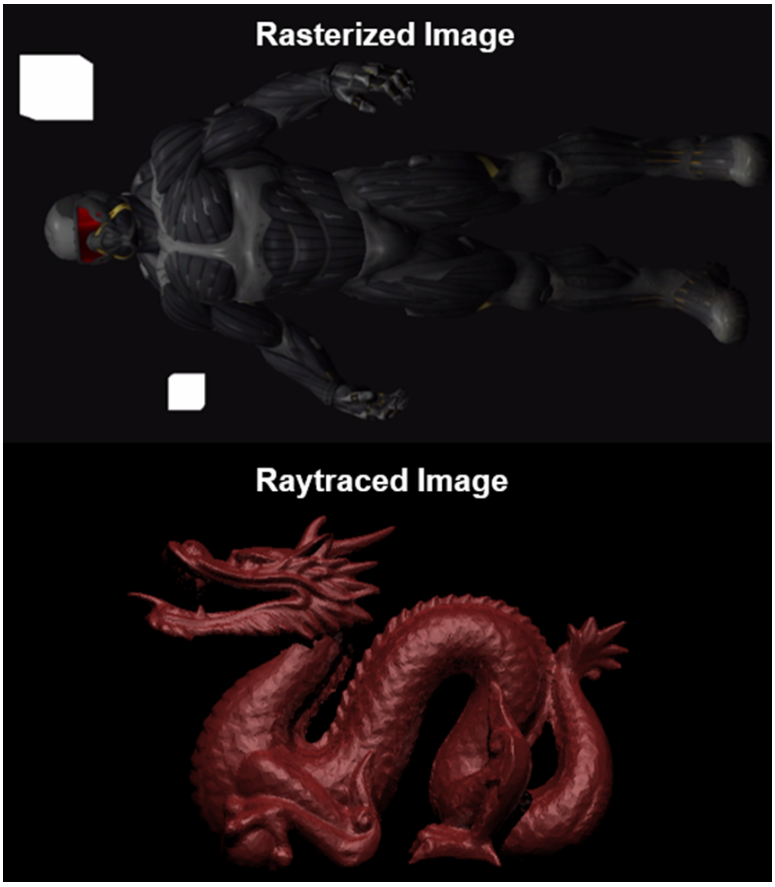

Figure 4: Comparison of the Rendered Image Quality of Rasterization Vs Raytracing

2.2 Acceleration Structures

CUDA enabled parallel implementation of raytracing mainly suffers from redundant ray-surface intersection. Acceleration structures are used to overcome this issue. Here scene objects are grouped into bins using respective algorithms and ray-surface intersection is checked for that bin first. If the ray doesn't hit the bin, then the ray-surface intersection for the objects found within the group can be skipped. In this way redundant ray-surface intersections can be reduced. The advancement in the construction and traversal of acceleration structure is the utmost factor which advances the computation intensive ray tracing rendering towards real time.

Acceleration structures are primarily categorised into spatial subdivision structure and object subdivision structure. And based on the level of implementation it's both these structures again categorised into flat and hierarchical as shown in 5 and 6.

The design of acceleration systems is usually done for static and offline rendering applications by an instruction processor instead of ray tracing acceleration hardware [46][45][39]. However, a strong need for online development and updating of the acceleration structure is demanded by the growing applications of rendering dynamic scenes. Recent breakthroughs on totally data-parallel construction algorithms also make it possible to design effective con-

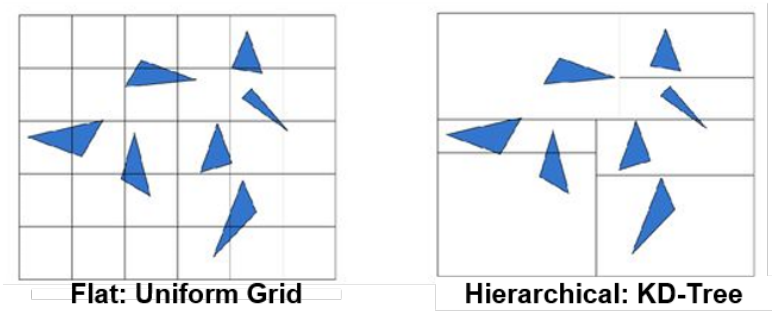

Figure 5: Spatial Subdivision Acceleration Structures

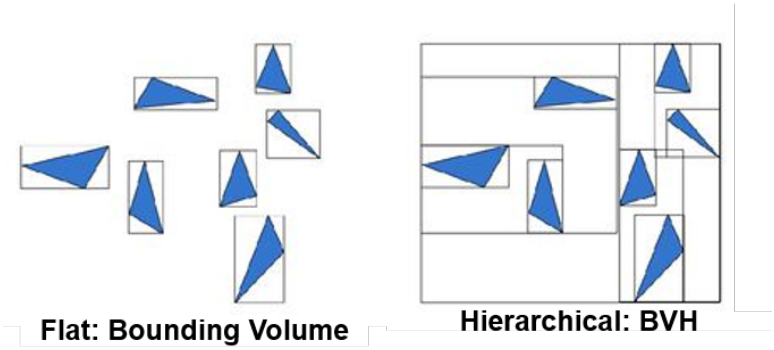

Figure 6: Object Subdivision Acceleration Structures

struction hardware [27][33]. Vaidyanathan et al. [50] performed a comprehensive analysis on the ray-tracing accuracy criterion and suggested a reduced accuracy approach that can be incorporated into current GPUs. Proposed ray tracer by Li et al. [32] implements an algorithm of highly efficient, completely parallel construction that applies to both Kd-tree and BVH as well as a leading-edge ray traversal process.

2.3 Uniform Grid

It's a flat spatial subdivision acceleration structure. Here a spatial point lies in a single node whereas an object might lie in multiple nodes.

The first Uniform Grid ray traversal algorithm was introduced by Fujimoto and Iwata[15] known as the 3D Digital Differential Analyser (3D-DDA) is an extension of Bresenham's algorithm which is frequently used for line rasterization. However, they introduced the 3D-DDA for traversing an Octree instead. In fact, each subdivision of Octree can be represented as a 2x2x2 cell Uniform Grid. The larger axis is defined by Bresenham's algorithm as the driving axis, determining the step size for the other (passive) axis. The key distinction is that all the intersected cells must be visited by the ray traversal algorithm, while certain cells from the passive axis can be skipped by the Bresenham's. The simplest acceleration structure is definitely a grid that regularly divides the space[15]. However, such a straightforward solution does not guarantee usefulness for scenes with an uneven distribution of

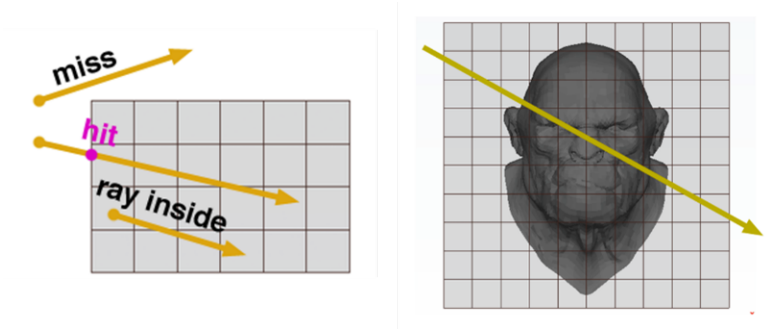

Figure 7: Uniform Grid

primitives, since it is possible to assign a vast number of primitives into a single cell. Amanatides and Woo [4] have proposed a 3D-DDA extension. It's the most common Uniform Grid traversal found in literature because of its easy and successful implementation.

2.4 Kd-tree

It's a hierarchical spatial subdivision acceleration structure. Here a spatial point lies in a single node whereas an object might lie in multiple nodes.

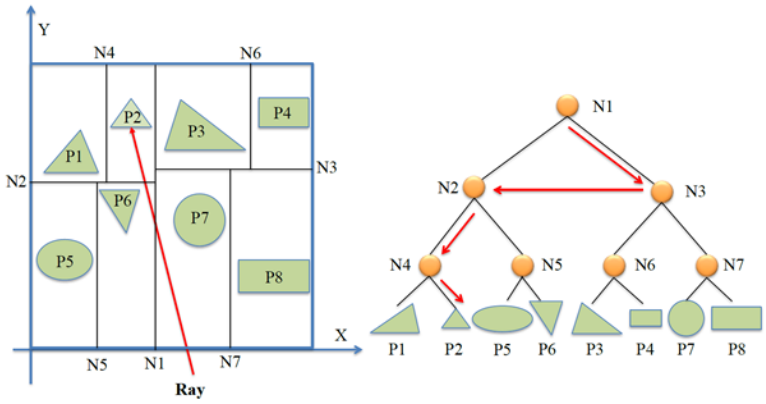

Figure 8: Kd-tree

Kd-tree, or k-dimensional tree, was initially introduced as a space partitioning data structure by Bentley [8] for indexing points in k-dimensional space. Kd-tree [14]is a special case of the binary space partitioning tree. The first established Kd-tree ray traverse algorithm was implemented by Kaplan [26], later referred to as Sequential Traversal [21]. Inside the Kd-tree nodes, this traverse performs a cyclic top-down point search, before the leaf containing the current search point is located. The point of interest is modified to be within the next leaf node after the leaf traversal and the search algorithm goes back to the root node. Many internal nodes are thus accessed repeatedly.

For Kd-tree, Jansen [25] suggested a recursive traversal algorithm. Since the recursive calls manage the ordering of nodes to be accessed, his strategy does not visit a node more

than once. It is constructed along a given axis by recursively splitting the space into two half-spaces. The cut location is chosen in such a way that there are approximately equal numbers of objects in the two halves. Typically, as used for ray tracing, it is not the recommended technique to allocate two half-spaces to an equal number of primitives when cutting a space. The number of primitives and the chance of a random ray entering any half of space is a safer heuristic to consider. In reality, this is the fundamental concept of Surface Area Heuristic (SAH) [18][36].

Via statistical analysis of ray-node intersection instances, Havran et al. [23] updated Jansen's traversal algorithm, minimizing traversal expense for more likely traversal situations. In addition, this traversal results in fewer numerical errors and therefore, fewer visual artifacts. Havran et al. [22] defined how ropes are used to build and traverse a Kd-tree. Ropes for Octrees and BSP-Trees are the neighbor-links between a leaf and its neighboring nodes. These connections can then be used by a traversal algorithm to directly access neighboring nodes, minimizing the number of internal nodes visited. This system is no longer a tree in the strict sense since it has cyclic ties, a result of more than one possible path from certain nodes to others. Havran [21] introduced an iterative version aiming efficiency, later called Kd-Standard Traversal [13]. To store the child node placed most distantly, the Kd-Standard traversal uses a stack. Later, after all the near-child nodes have been visited, each stacked node is visited. The stack implies, thus, that the traversal takes place in the same order as the recursive solution.

Via statistical analysis of ray-node intersection instances, Havran et al. [23] updated Jansen's traversal algorithm, minimizing traversal expense for more likely traversal situations. In addition, this traversal results in fewer numerical errors and therefore, fewer visual artifacts. Havran et al. [22] defined how ropes are used to build and traverse a Kd-tree. Ropes for Octrees and BSP-Trees are the neighbor-links between a leaf and its neighboring nodes. These connections can then be used by a traversal algorithm to directly access neighboring nodes, minimizing the number of internal nodes visited. This system is no longer a tree in the strict sense since it has cyclic ties, a result of more than one possible path from certain nodes to others. Havran [21] introduced an iterative version aiming efficiency, later called Kd-Standard Traversal [13]. To store the child node placed most distantly, the Kd-Standard traversal uses a stack. Later, after all the near-child nodes have been visited, each stacked node is visited. The stack implies, thus, that the traversal takes place in the same order as the recursive solution.

Some improvements to the Kd-Restart were suggested by Horn et al. [24]in order to minimize the number of revisited nodes. Their Push-Down algorithm switches the node of the root search to the last one where only one of its children is hit by the ray. As the other child will never be visited, this internal node securely becomes the new root search node. Then when a restart event is triggered, the search returns to this node instead of the tree's root node. They also described the algorithm of short-stack, like a hybridization of the traversals of Kd-Standard and Kd-Restart. The short-stack uses a small circular array representing a short stack instead of a wide, tree-depth-sized stack, with a length that takes into account the hardware architecture's resource constraints. If the stack is not empty, the next node to be accessed is popped out of the stack on a short-stack traversal, equivalent to the Kd-standard traversal, thereby preventing re-visiting any nodes. The restart case is otherwise processed in the same manner as Kd-Restart. Finally, to the awareness of

the author, Horn's implementation of GPU ray tracing is the first one to reach interactive speeds.

A CUDA Kd-tree packet ray traversal based on the stackless algorithm of ropes, displaying interactive frame rates, was proposed by Popov et al.[42]. The stackless existence of a rope traversal is worth noting, as the leaves' neighbor-links are adequate to locate the next tree branch to be traversed. A stackless algorithm on a GPU minimizes high latency memory bandwidth usage. For modern GPU Kd-tree implementations, a ropes-based traversal algorithm is therefore an important reference. Kd-tree acceleration structure of Hapala and Havran[20], leveraging ray coherence and algorithms built with unique hardware architecture limitations such as memory latency and ray re-order consumption. Many strategies have been developed [48]to enhance the coherence of parallel ray tracing activities. The main concept is to group rays with similar traversal characteristics and then issue the memory requests in parallel.

Recently, Li et al. [33] also suggested a completely parallel construction algorithm. In this work, a maximum degree of parallelism is allowed by starting from a Morton code [37] based ordering of primitives and concurrently working on all leaf nodes or internal nodes to form a hierarchy in a bottom-up manner. The construction algorithm based on binning produces a small loss in the quality of the resulting Kd-trees [38]. Liu et al. [35] suggested a FastTree hardware accelerator to fix this issue and also allow for an even higher construction throughput. The completely parallel construction algorithm suggested by Li et al. [33] is adopted by the FastTree hardware. The construction algorithm begins with a conceptually complete binary tree, which corresponds to a standard space division into grid cells, and then operates bottom-up on all leaf nodes at the same time. A benefit of the algorithm is that a group of prefix-sum and radix sort units can be mapped to the main operations, which Liu et al. [35] implement successfully into hardware.

2.5 BVH

It's a hierarchical object subdivision acceleration structure. Here an object lies in a single node whereas a spatial point might lie in multiple nodes.

Clark [10] first proposed the BVH. The BVH suggested by Rubin and Whitted[44] could be the oldest AS for ray tracing, or at least some of its principles, such as close bounding volumes for termination of early ray object intersection. A top-down BVH traversal algorithm was proposed by Kay and Kajiya [29], in which the ray is evaluated against both bounding volumes from the child nodes. The closest child node from the origin of the ray is accessed while the farthest node is stored for a future traversal in a priority queue like a heap. The authors did not, however, display any output gains than a simple stack. Shirley and Morley [49] have actually warned about the detrimental effects of a priority queue overhead.

Arvo and Kirk's [6] works on BVH construction usually pursue a bottom-up approach by hierarchical triangle clustering. A common situation is where a static background and several dynamic objects make up the scene. The concept of two-level BVHs was proposed by Wald et al [52]. A top BVH is constructed over the individual objects and each object has its BVH. More accurately, the top BVH includes references to individual objects with

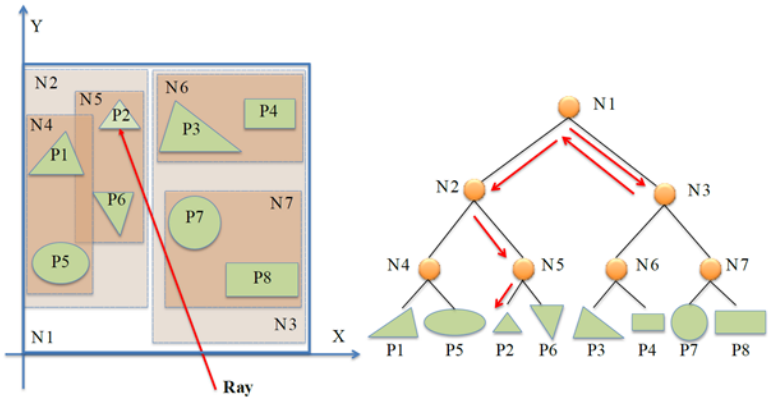

Figure 9: BVH

local transformations that help rigid body animations and instancing. A caveat of the two-level BVH is that if the individual objects intersect, it could create a traversal overhead. Benthin et al [7] recently addressed this issue.

Recent works are based on the binning system suggested by Wald [51]to increase the speed of traversal. The main concept is to delegate primitives to frequently positioned bins and then to find the better cut at the bins' boundary. The right cut is measured in terms of SAH costs here. This BVH construction algorithm powered by SAH consists of both a parallel vertical and a horizontal point. The former stage deploys one thread starting from the root to operate on the top-most subtree, while the latter stage uses several threads to manage subtrees descending from the top subtree simultaneously. Available parallelism is maximized through such an arrangement.

As separate rays can follow different execution routes, the traversal algorithm does not work well with the single instruction, multiple data (SIMD) hardware of modern GPUs. Merging the traversal and intersection test codes into a single kernel [1] and launching a wide number of concurrent threads is an efficient practice for GPU-based ray tracers. BVH with Spatial Splits (SBVH) was introduced by Stich et al. [47]. An algorithm that permitted stackless traversals on BVHs was proposed by Laine [30]. The cost of the revisited nodes on existing GPU architectures overshadows the gains of these algorithms. Laine indicates, however, that this strategy may perhaps be useful in other and future architectures by eliminating problems such as cache trashing.

Most BVH developers today focus on the bin-based algorithm in which the Moron code is used as a clustering basis.[31][16]. Pantaleoni and Luebke [40] systematically enhanced BVH construction efficiency by incorporating appropriate procedures to deal with the Moron code and taking advantage of the primitives' inherent coherence.

An algorithm that permitted stackless traversals on BVHs was proposed by Hapala et al. [19]. Aila et al. [2] recommended the use of persistent threads controlling a kernel to significantly minimize the number of idle threads in a CUDA block in order to improve the performance of ray traversals over the CUDA architecture. These persistent threads offer

major performance improvement and can be combined with any AS. It requires a processing flow that is top-down. Primitives belonging to a node are split into a specified number of bins along each axis at each stage and pick up the best direction and position for a cut. Again, there are many metrics for determining a cut's efficacy. SAH is generally considered the best among these metrics, but a few recent works have managed to extract good partition efficiency to escape the comparatively costly heuristic and focus solely on Morton code.

The concept of reconstructing an existing low-quality BVH by operating simultaneously on local tree structures was suggested by Karras and Aila [28]. The principle is capable of massively parallel processing and has been proven to be incredibly effective. Comprehensive research on the precision requirement of ray tracing was carried out by Liktor and Vaidyanathan [34] and suggested a reduced precision approach that can be incorporated into current GPUs. Ylitie et al. [54] recently recommended that compressed 8-ary BVHs be used to achieve the best efficiency.

3 Problem Statement

The sequential implementation of raytracing in CPU benefits from large main memory to handle recursive calls in contrast to the parallel implementation of raytracing in GPU which suffers from limited shared memory. This results in low quality rendered images. And furthermore redundant ray-surface intersection restricts the raytracing implementation from achieving real time rendering. This paper tries to implement a CUDA accelerated raytracer which render high quality images in real time. Since raytracing imageries are mainly used in entertainment and medical field, the quality of the rendered image and time taken to render the image are crucial.

4 Proposed Solution

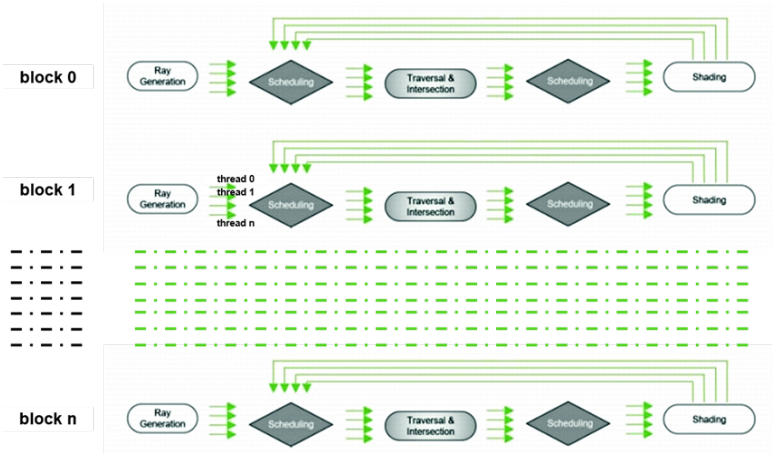

Figure 10: CUDA enabled SIMD Parallel Architecture of raytracer

One significant feature of raytracing algorithms is that it is possible to calculate the final color of each pixel in the image independently of the others, providing a very natural way to parallelize the execution of a raytracer by separating pixel color measurements between threads. This is an ideal situation to deploy Single Instruction Multiple Data (SIMD) parallel architecture. The functional block diagram in Figure 10 shows the implemented SIMD architecture for this project.

To address the stack overflow issue, recursive calls to the functions are converted to iterative calls with explicitly defined stack depth. The c++ code snippets of recursive and iterative calls for pixel colour calculation known as shading are given below.

```
1  //recursive call
2  color shade(const ray& r, const hittable& world, int depth) {
3      hit_record rec;
4      if (depth <= 0)
5          return color(0,0,0);
6      if (world.hit(r, 0.001, infinity, rec)) {
7          ray scattered;
8          color attenuation;
9          if (rec.mat_ptr->scatter(r, rec, attenuation, scattered))
10             return attenuation * shade(scattered, world, depth-1);
11         return color(0,0,0);
12     }
13     return color(0,0,0);
14 }
15
16 //iterative call
17 color shade(const ray& r, const hittable& world, int depth) {
18     ray cur_ray = r;
19     vec3 cur_attenuation = vec3(0.0,0.0,0.0);
20     for(int i = 0; i < depth; i++) {
21       hit_record rec;
22   if (world.hit(cur_ray, 0.001, infinity, rec)) {
23           ray scattered;
24           color attenuation;
25           if (rec.mat_ptr->scatter(cur_ray, rec, attenuation, scattered)) {
26                cur_attenuation *= attenuation;
27    cur_ray = scattered ;
28   }
29   else
30               return color(0,0,0);
31   }
32   else
33   return cur_attenuation;
34     }
35  return cur_attenuation;
36 }
```

The iterative call carefully considers the different modes of shading to account for the depth. Figure 11 shows the rendered images of different shading modes.

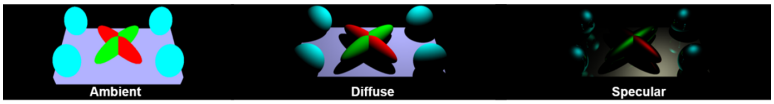

Figure 11: Different Modes Shading in Raytracing

To address the redundant ray-surface intersection, this paper implements Binary Volume Hierarchy (BVH) acceleration structure. When each primitive is read by the raytracer, a bounding volume is attached to it and compound to form the outer bounding volume. In this way BVH is generated in the CUDA kernel. And when ray-surface intersection happens, this binary volume hierarchy is traversed in a top-down order and identifies the correct hit in log(n) times. A huge number of redundant ray-surface intersections is omitted through the binary tree traversal.

5 Experimental Evaluation

The following results were obtained when comparing the sequential implementation in Intel i7-4710HQ 4 core CPU @ 2.5 GHz versus the parallel implementation in the NVIDIA GeForce 840M GPU. Both the implementations were tested in the Ubuntu 18.04 Linux Environment. For each comparison physical optical properties such as reflection, refraction, defocus blur and motion blur were implemented and rendered into an HD image (resolution 1280x720) with 100 samples (to avoid anti-aliasing) & 50 depth (number of generated secondary rays per primary rays) per pixel which generated 1280x720x100 = 92,160,000 (92 million) primary rays and 92,160,000x50 = 4,608,000,000 (4.6 billion) secondary rays.

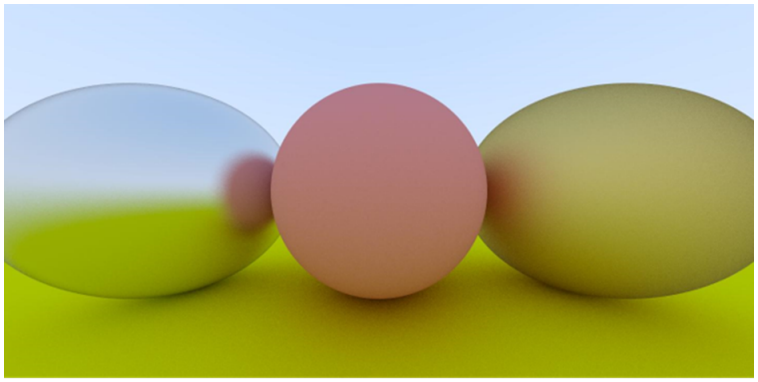

Figure 12: Reflection

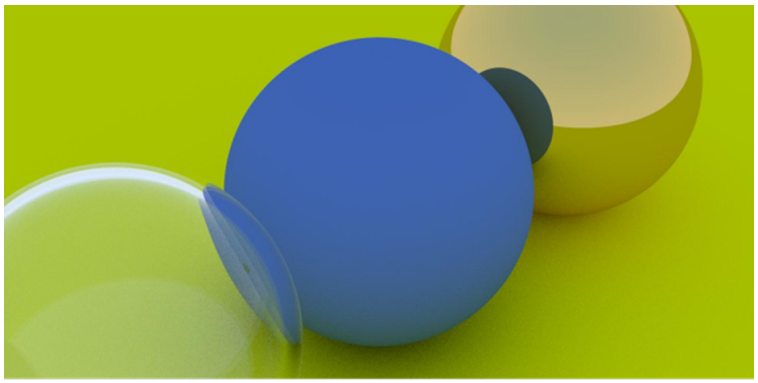

Figure 13: Refraction

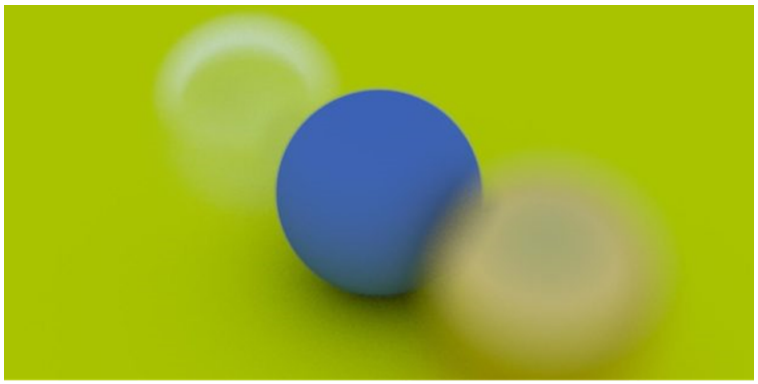

Figure 14: Defocus Blur

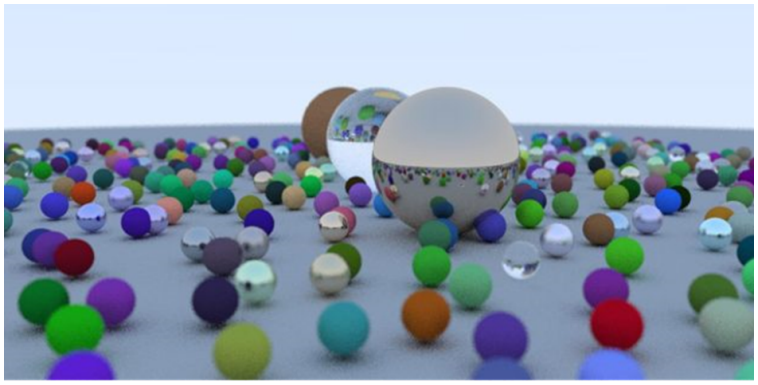

Figure 15: Motion Blur

6 Conclusion

While the rendered optical property increases in complexity as in the order of reflection, refraction and defocus blur, the acceleration ratio of GPU vs CPU also increases in the order of x2.472, x2.942, x3.087. This shows parallel implementation of raytracing algorithm using BVH acceleration structure is better suited to achieve the interactive speeds needed for the real time rendering. In addition, when the rendered number of objects in the scene increases, the acceleration ratio of GPU vs CPU also increases from around x3 to x21. This shows again the implemented raytracing engine gives a cutting-edge advantage for a scene with large number of objects.

This raytracing rendering performance is measured in the author's laptop NVIDIA GeForce 840M GPU card, which came to the market in 2014 with only 384 CUDA cores. Currently this is an outdated GPU when compared to the latest NVIDIA GeForce RTX 3090 GPU card which came to market in September 2020 with 10496 CUDA cores. If the implemented parallel raytracing rendering engine is deployed in the commercially available state of the art GPUs, it can render high quality images in real time needed for entertainment and medical field.

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149, 2009.

[2] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[3] Tomas Akenine-Mo, Eric Haines, Naty Hoffman, et al. Real-time rendering. 2018.

[4] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[5] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, 1968.

[6] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. *An introduction to ray tracing*, pages 201–262, 1989.

[7] Carsten Benthin, Sven Woop, Ingo Wald, and Attila T Áfra. Improved two-level bvhs using partial re-braiding. In *Proceedings of High Performance Graphics*, pages 1–8. 2017.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] Per H Christensen, Julian Fong, David M Laur, and Dana Batali. Ray tracing for the moviecars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6. IEEE, 2006.

[10] James H Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[11] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys (CSUR)*, 50(4):1–41, 2017.

[12] Andreas Dietrich, Abe Stephens, and Ingo Wald. Exploring a boeing 777: Ray tracing large-scale cad data. *IEEE Computer Graphics and Applications*, (6):36–46, 2007.

[13] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[14] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, 1980.

[15] Akira Fujimoto and Kansei Iwata. Accelerated ray tracing. In *Computer graphics*, pages 41–65. Springer, 1985.

[16] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 59–64, 2011.

[17] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[18] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[19] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, pages 7–12, 2011.

[20] Michal Hapala and Vlastimil Havran. Kd-tree traversal algorithms for ray tracing. In *Computer Graphics Forum*, volume 30, pages 199–213. Wiley Online Library, 2011.

[21] Vlastimil Havran. *Heuristic ray shooting algorithms*. PhD thesis, Ph. d. thesis, Department of Computer Science and Engineering, Faculty of . . . , 2000.

[22] Vlastimil Havran, Jirí Bittner, and Jirí Zára. Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics*, pages 130–140. Citeseer, 1998.

[23] Vlastimil Havran, Tomas Kopal, Jiří Bittner, and Jiří Žára. Fast robust bsp tree traversal algorithm for ray tracing. *Journal of graphics tools*, 2(4):15–23, 1997.

[24] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, 2007.

[25] Frederik W Jansen. Data structures for ray tracing. In *Data Structures for Raster Graphics*, pages 57–73. Springer, 1986.

[26] Michael R Kaplan. Space-tracing: A constant time ray-tracer. In *SIGGRAPH'85 State of the Art in Image Synthesis seminar notes*, volume 18, pages 149–158, 1985.

[27] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37, 2012.

[28] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99, 2013.

[29] Timothy L Kay and James T Kajiya. Ray tracing complex scenes. *ACM SIGGRAPH computer graphics*, 20(4):269–278, 1986.

[30] Samuli Laine. Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics*, pages 107–111. Citeseer, 2010.

[31] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[32] Zonghui Li, Yangdong Deng, and Ming Gu. Path compression kd-trees with multi-layer parallel construction a case study on ray tracing. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–8, 2017.

[33] Zonghui Li, Tong Wang, and Yangdong Deng. Fully parallel kd-tree construction for real-time ray tracing. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 159–159, 2014.

[34] Gábor Liktor and Karthikeyan Vaidyanathan. Bandwidth-efficient bvh layout for incremental hardware traversal. In *High Performance Graphics*, pages 51–61, 2016.

[35] Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. Fasttree: A hardware kd-tree construction acceleration engine for real-time ray tracing. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1595–1598. IEEE, 2015.

[36] J David MacDonald and Kellogg S Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[37] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[38] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics (TOG)*, 33(5):1–15, 2014.

[39] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. T&i engine: traversal and intersection engine for hardware accelerated ray tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, pages 1–10, 2011.

[40] Jacopo Pantaleoni and David Luebke. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95, 2010.

[41] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[42] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Wiley Online Library, 2007.

[43] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, pages 268–es. 2005.

[44] Steven M Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, 1980.

[45] Jörg Schmittler. Saarcor: a hardware architecture for real-time ray tracing. 2006.

[46] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor: a hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, 2002.

[47] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, 2009.

[48] Wang Tong and Yangdong Deng. Mining effective parallelism from hidden coherence for gpu based path tracing. In *SIGGRAPH Asia 2013 Technical Briefs*, pages 1–4. 2013.

[49] Realistic Ray Tracing. /p. shirley, rk morley.—wellesley: Ak peters, 2003.

[50] Karthikeyan Vaidyanathan, Tomas Akenine-Möller, and Marco Salvi. Watertight ray traversal with reduced precision. In *High Performance Graphics*, pages 33–40, 2016.

[51] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40. IEEE, 2007.

[52] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.*, pages 77–85. IEEE, 2003.

[53] Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, pages 4–es. 2005.

[54] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics*, pages 1–13. 2017.