Diagnosing Limitations of and Evaluating Ray Tracing Acceleration Paradigms on Integrated Platforms

Siraaj Sandhu

ABSTRACT

This paper evaluates existing ray tracing acceleration paradigms on integrated platforms, such as the iGPUs found in many mobile devices and laptops. The ray tracing algorithm has become more popular in industry and academia in recent years given numerous hardware advancements, but development and comprehensive comparison of optimization algorithms is incongruous across platforms and especially lacking on integrated GPUs. We select five representative algorithms that all pertain to any one of three acceleration paradigms, each of which is characterized by its own design philosophy for accelerating the ray tracing algorithm. These algorithms are then benchmarked in HD resolution on Intel's integrated graphics architecture using three test models and three different vantage points per model. The results contradict the existing consensus that has been developed for dedicated graphics cards and offer three factors, one novel, with particularly profound implications for paradigm performance: cache coherency, thread utilization, and energy usage. While the resultant ranking suggests the Uniform Grid paradigm outperforms the BVH and kd-tree paradigms on integrated platforms, it also offers insight into areas of improvement and future design considerations.

1. INTRODUCTION

[1.1] Ray Tracing

Various applications of computer graphics necessitate photorealistic imagery, including, but not limited to: product, architectural, and industrial CAD; simulation of optical and lighting systems; and most recently, in CGI and VFX for popular media such as films and video games. The ray tracing algorithm uses rays to "shade" a 3D scene and approximate its chiaroscuro (Appel et al, 1968). When the "rendering equation," a geometrical optics approximation, is used as a shading model, ray tracing is capable of replicating a variety of optical phenomena, such as scattering, reflection, refraction, soft shadows, caustics, depth-of-field, and motion blur (Kajiya et al, 1986). The tradeoff for accuracy is time complexity. 3D models are typically composed of triangle *primitives*, so an intersection must be found between each ray and a triangle from the scene. Intersection tests are costly, so the naive linear-search approach scales poorly as 3D models grow in size, usually into the millions of triangles.

[1.2] Optimization & Acceleration Paradigms

Since its inception, attempts to improve ray tracing performance have been well-researched, typically operating on the assumption that rays will not intersect every triangle in a scene, and consequently, that minimizing the number of intersection tests will result in sub-linear intersection time (Kay et al, 1968). These methods, or *acceleration structures*, typically create a hierarchy or tree to achieve logarithmic traversal time. These hierarchies can either subdivide the set of scene primitives, as is typically done by *bounding volume hierarchies* (BVH) (Clark et al, 1976), or space itself, as is done by *kd-trees* (Fussell et al, 1988). Alternatively, non-hierarchical structures subdivide space into non-overlapping volumes, as is done by *uniform grids* (Fujimoto et al, 1986). These three classes of algorithms and their accompanying data structures encompass most existing acceleration schemes, so we will treat them as *acceleration paradigms*.

[1.3] Performance Comparison of Acceleration Paradigms

All paradigms have the same object–that is, to minimize the number of intersection tests performed–so the question regarding which paradigm will produce the fastest trace time naturally arises. Since the ray tracing algorithm entails independently tracing millions of rays, it is not only notoriously hardware-limited but also suited for a parallelized implementation on the GPU, where it benefits from *same-instruction-multiple-thread* (SIMT) and *same-instruction-multiple-data* (SIMD) architectures. Consequently, extensive performance comparisons have been conducted, formerly on CPUs and more recently on dedicated GPUs. Vinkler et al (2015) found that on the NVIDIA Kepler architecture, BVH performed better than kd-trees for scenes of trivial to moderate complexity, though kd-trees proved especially performant on highly-complex scenes. The study used only stack-based traversal schemes for BVH, though stackless algorithms exist. Meister et al (2022) addressed the lack of comprehensive performance comparisons of different types of bounding volume hierarchies by conducting an empirical comparison of prominent algorithms that fall under the BVH paradigm. It was found that a hybrid BVH-kd algorithm performed the best on an AMD Radeon series graphics card. Aila et al (2009) discussed the adaptation of BVH algorithms to contemporary NVIDIA hardware. The difference between theoretical results and hardware benchmarks was examined and hypotheses regarding failure to meet theoretical expectations were presented, particularly as it related to suboptimal work distribution on the GPU. Based on these comparisons and decades of application in industry, BVH has become the preferred paradigm for efficient ray tracing (Wodniok et al, 2016). In fact, NVIDIA has begun implementing dedicated hardware acceleration for BVH construction and traversal in their RTX series of GPUs.

[1.4] Integrated Platforms

However, the consensus discussed in [1.3] is only applicable to dedicated graphics cards, such as those produced by AMD and NVIDIA, and is not generalizable to other platforms, such as integrated GPUs (iGPUs), due to inherent hardware differences. Firstly, iGPUs are power- and heat-limited, as they are most commonly used on low-power devices such as mobile phones and laptops. As such, iGPUs will throttle, or lower clock speeds should a workload prove too computationally demanding. Without dedicated VRAM, iGPUs must share memory with the CPU and are consequently limited by smaller memory, smaller caches, and less bandwidth compared to dedicated graphics cards. The variation in these parameters means existing benchmarks cannot accurately predict the performance of acceleration paradigms on integrated platforms. Yet there is both academic and industrial interest in ray tracing on integrated platforms. Seo et al (2022) examined the untapped computing potential of phones and laptops and proposed a distributed ray tracing algorithm. Their results suggest that clustered computing offers a way to carry out large-scale rendering and scientific computing tasks in an energy-efficient manner. Andrade et al (2014) demonstrated that ray tracing can be used to supplement alternative rendering paradigms to improve graphical fidelity. The most immediate application of their and others' hybrid methods is in video games: improved graphical fidelity for marginal performance costs can improve the accessibility and marketability of games targeted for low-power platforms. As such, the question regarding which acceleration paradigm performs best on iGPUs is relevant for both aforementioned applications. This paper will attempt to answer this question by presenting a comprehensive performance comparison for the three aforementioned acceleration paradigms by implementing algorithms pertaining to each class from representative papers. Hypotheses for any discrepancies and suggestions for further development of these algorithms will also be proposed.

2. PREVIOUS WORK

[2.1] BVH

Clark et al (1976) were one of the first to present the *bounding volume hierarchy* (BVH). The structure is a specialized binary search tree: first, all triangle primitives are wrapped in tight *axis-aligned bounding boxes* (AABB), after which they are partitioned into a left subgroup and a right subgroup according to some criteria

or heuristic. This subdivision is repeated recursively until all groups contain only one triangle, i.e. the leaf nodes of the tree have been created. Each node in the tree encloses its children with its own AABB. Traversal of the BVH is then just a traversal of a binary search tree. At each node, should the ray fail to intersect its bounding box, then exploration of that node's subtree is unnecessary. This takes the form of a depth-first search, as detailed in Algorithm 1:

Algorithm 1. Typical stack-based BVH traversal

```
func intersect(ray, bvh):
  push bvh.root onto stack
  while stack is not empty:
    pop node from stack
    if ray intersects node.bounding_box:
      if node is not leaf:
        push node.right, node.left onto stack
      else:
        test ray intersection with node.triangle
```

Ray-box intersections are typically much cheaper than ray-triangle intersections. Then, traversal complexity follows that of a binary tree: $O(\log n)$ on average and $O(n)$ in the worst case, if the tree regresses into a linked list, for instance. Thus BVH performance is heavily dependent on construction and traversal order.

Firstly, the question as to how triangles should be subdivided must be answered. The state-of-the-art method extends Goldsmith et al's *surface area heuristic* (SAH) (1987). Intuitively, any subset of primitives can be partitioned at any index and the SAH simply assigns a cost to each choice of partition. A greedy algorithm can then minimize this cost during tree construction. Suppose we are considering a partition A and B of a given subset of primitives S such that $A \cup B = S$ and $A \cap B = \{\}$. The cost heuristic

$$c_{SAH}(A,B) = p_A \sum_{i=1}^{|A|} c_{isect}(A_i) + p_B \sum_{i=1}^{|B|} c_{isect}(B_i)$$

offers a model to measure the cost of this partition, where $c_{isect}(p)$ is the cost of intersecting a triangle primitive (we will assume it is constant), $p_A = P(A|S)$, and $p_B = P(B|S)$. This heuristic essentially computes the sum of the linear traversal costs of the left and right partitions, weighted by the conditional probability that any ray will intersect that partition's bounding box given it has intersected its parent's bounding box. For any uniformly distributed random ray, the probability that it will enter the bounding volume of A given it has intersected the bounding volume of its parent, S, is $P(A|S) = \frac{s_S}{s_A}$, where s_Σ is the surface area of the volume pertaining to the set of primitives Σ, hence the surface area heuristic. A tree built using the SAH will demonstrate higher performance during traversal, so it is critical that all trees built for this benchmark use the SAH to ensure a fair comparison.

The second factor impacting BVH performance is traversal order. When intersecting rays with scenes, only the triangle closest to the ray's origin is of interest, as this is the only "visible" triangle, so intersections with farther triangles are ultimately unnecessary. It is thus advantageous to traverse nodes in a front-to-back order by visiting the node closer to the ray's origin first. If maximum distance constraints are imposed on ray-box intersections, extraneous visits to subtrees of farther nodes can be avoided. Dammertz et al (2008) present a heuristic for ordering child nodes in a front-to-back manner based on the split-axis along which they were partitioned and the sign of the ray's direction along that axis. For this benchmark, we will deem this heuristic the *split-axis heuristic*. Wald et al (2014) present a heuristic for ordering child nodes in a front-to-back manner based on the computed distance from the ray's origin to each child node. When any node is visited, Wald et al's method, which we will deem the *distance heuristic*, entails intersecting the ray with the bounding volumes of both of the node's children at the same time and comparing the resultant distances.

Alternatively, it is possible to opt for a stackless BVH traversal scheme. Smits et al (1998) proposed that the need to maintain a traversal stack could be eliminated by storing *miss links* in each node, which point to the next adjacent subtree. Should the ray fail to intersect a node's bounding volume, the traversal algorithm can follow the miss link to bypass that node's subtree. Typically, a miss link simply points to a node's sibling or the next node in a depth-first search (DFS) of the tree. Otherwise, failure to intersect the node's bounding volume would force traversal to terminate. This algorithm ensures constant space complexity with the tradeoff being that fixed traversal order is enforced, leading to extraneous intersection tests. It will be valuable to benchmark all of the aforementioned methods because of the wide variety in

existing BVH designs and because it is unpredictable how each parameter will interact with memory on iGPUs

[2.2] kd-tree

The kd-tree is structurally similar to the BVH, as it is also a specialization of a binary tree. Fussell et al (1988) first discussed the application of the kd-tree data structure to ray tracing, which partitions space instead of the triangle primitives themselves. As such, triangles can belong to one or both of the left and right partitions at any given node in the tree. Consequently, there is a potential duplication of information in the tree, but this can prove advantageous because traversing nodes in a kd-tree is computationally faster than traversing nodes in a BVH, which requires a ray-box intersection test to be conducted. Rather, algorithm 2 presents the typical kd-tree traversal scheme:

Algorithm 2: Typical stack-based kd-tree traversal

```
func intersect(ray, kd_tree):
  test ray intersection with root bounding box
  push kd_tree.root onto stack
  while stack is not empty:
    pop node from stack
    update search window
    if node is not leaf:
      compute parametric distance at which ray
        intersects the split plane
      order node.children based on ray direction and
        parametric distance
      push child nodes in front-to-back order onto
        stack as necessary
    else:
      foreach triangle in node.triangles:
        test ray intersection with triangle
```

The kd-tree traversal algorithm does not require bounding box intersections, and finding the parametric distance at which a ray intersects any particular split plane given it intersects the global bounding volume amounts to solving a problem of similar triangles and computing a single ratio. Front-to-back traversal order is also built into the typical kd-tree traversal scheme, as child nodes must be ordered before pushing them onto the stack. The parametric distances are used to determine whether the ray intersects any given child at all and sort relevant children according to front-to-back order before pushing them onto the stack.

Foley et al (2005) present the *kd-restart* algorithm which eliminates the need to maintain an external traversal stack by "restarting" traversal from the root with an updated search window so that visited nodes are immediately skipped. The *kd-restart* algorithm is better suited for implementation on GPUs because of its constant space complexity. As such we intend to use Foley et al's modification for this benchmark. Moreover, kd-tree space partitions will also be determined using the surface area heuristic, much like the BVH. Due to the similarities in structure with the BVH, the time complexity of the kd-tree exhibits the same average and worst cases as the BVH.

[2.3] Uniform Grid

Fujimoto et al's uniform grid (1986) offers a non-hierarchical way to traverse a scene in sublinear time. A bounding box is created for the entire scene and subsequently divided into uniformly sized cells. Each cell is assigned a set of triangle primitives that overlap it. Traversal first tests whether the ray intersects the global bounding volume, and if it does, progressively "marches" the ray through the grid by advancing the ray from cell to cell until it exits the grid. Only triangles in cells that the ray intersects need to be tested for intersections. Via Fujimoto et al's proposed *Three Dimensional Digital Differential Analyzer* (3DDDA) algorithm, subsequent cells are highly predictable and entirely dependent on the cell at which the ray entered the grid and the direction of the ray. The 3DDDA algorithm thus presents a lightweight traversal algorithm as detailed in Algorithm 3:

Algorithm 3: 3DDDA

```
func intersect(ray, grid):
  test ray intersection with global bounding box
    and store parametric distances at which ray enters
    and exits the grid
  compute current_cell and exit_cell based on
    parametric distances
  while current_cell != exit_cell:
    foreach triangle in current_cell.triangles:
      test ray intersection with triangle
      update current_cell coordinates within grid to
        step the ray forward to the next cell
```

The uniform grid thus offers constant space complexity for its traversal. Time complexity is dependent on the manner in which the grid has been subdivided. Cleary et al (1983) present a heuristic for determining the size or resolution of the grid:

$$\vec{N} = \vec{B}\sqrt[3]{\frac{\lambda|P|}{V}}$$

Where $|P|$ is the number of triangle primitives in the grid, $\vec{B}$ stores the size of the global bounding box, V is the volume of the global bounding box, λ is a modifiable parameter, and $\vec{N}$ stores the resolution of the grid along each axis. For this benchmark, we will deem this the *cube-root heuristic* (CRH). We expect our usage of an established resolution heuristic in this benchmark to produce relatively optimal grids.

Generally, it is expected that the time complexity of the algorithm will be proportional to the number of cells that a ray intersects and the number of primitives that ray intersects in each cell. The number of cells will be proportional to the grid resolution and the number of primitives in each cell will on average be $\frac{n}{C}$, where C is the number of cells. Then we suggest the runtime complexity is

$$T_{grid}(n) \sim \sqrt[3]{n} \cdot \frac{n}{C} \sim \sqrt[3]{n} \cdot \frac{n}{(\sqrt[3]{n})^3} = \sqrt[3]{n} \in O(\sqrt[3]{n})$$

3. MATERIALS & METHODS

[3.1] Algorithm Selection & Implementation

For this benchmark, we will compare the performance of the five algorithms discussed in Previous Work. These include:

Table 1. Selected algorithms for benchmarking and the names by which they will be referred to in the results

- BVH (SAH-based construction)
  1. split-axis heuristic (`BVH_FTB_DF`)
  2. Distance heuristic (`BVH_FTB_BF`)
  3. Stackless (`BVH_STACKLESS_DF`)
- kd-tree (SAH-based construction)
  4. kd-restart (`KD_TREE`)
- Uniform Grid (CRH-based construction)
  5. 3DDDA (`GRID`)

These algorithms were chosen not only to verify which paradigm yields the fastest trace time but also to probe the specific limitations of iGPUs and see how these factors, such as limited memory and cache, will interact with varying time and space complexities and storage schemes of each algorithm.

Firstly, grids and hierarchical paradigms exhibit cube root and logarithmic time complexity, respectively. The benchmark will thus show whether theoretical runtime is an accurate predictor of actual performance.

Grids, kd-trees, and stackless BVH are characterized by constant space complexity while the other two BVH methods maintain stacks in their traversals. Integrated GPUs are both highly parallel and memory-limited. Supposing a fixed amount of memory is available for use at any given time, we expect algorithms with higher space complexity to occupy fewer threads at a time than constant-space algorithms, resulting in underutilization of available threads on the iGPU. It is therefore possible that constant-space algorithms could perform better.

All algorithms have different memory layouts. Nodes in kd-trees and cells in grids must store lists of triangle primitives. To ensure fixed-width cells, an auxiliary array stores all primitive IDs and each cell need only store an offset into the auxiliary array and a count indicating how many subsequent primitives belong to that cell. These parameters allow the traversal algorithm to reconstruct the primitive list for each grid cell by taking a subset of the auxiliary ID array. Figure 1 depicts this storage scheme. Similarly, all nodes in a kd-tree store lists of triangle primitives. An auxiliary array with the same form and function as the one used by the grid is thus employed for kd-tree traversal. The auxiliary array constitutes an extra layer of indirection between acceleration structure nodes and the primitives they contain, which can incur additional memory access overhead. However, traversal will still access elements sequentially starting at an offset, which is relatively

cache-friendly. Moreover, we inform our choice of λ for grid construction based on Wald et al's (2006) results, which allege that a lambda between 3 and 5 is sufficient to produce relatively optimal grid resolutions. As such we set our $\lambda = 3$ for all tests.

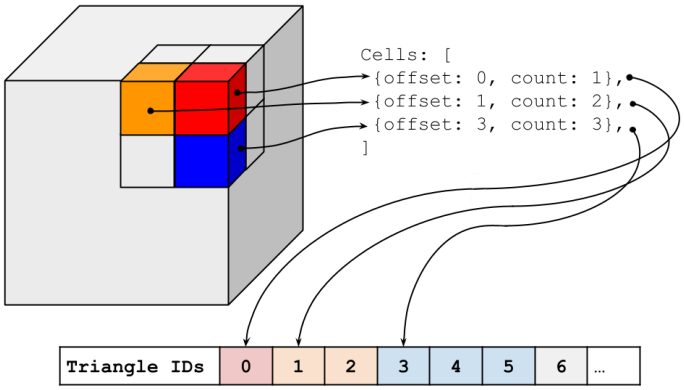

Fig. 1. Primitive list storage scheme for grid cells. Cells store an offset into an auxiliary array and a count indicating how many subsequent primitives belong to that cell. Elements in the auxiliary array have been colored to indicate what grid cell they belong to.

All BVH algorithms do not require auxiliary data structures. This benchmark may therefore indicate whether maintenance of an additional, auxiliary data structure has any major impact on cache utilization and performance. Leaf nodes directly store their respective primitive IDs. All BVH types additionally have the same memory footprint, though the space is utilized differently. The stackless and split-axis heuristic algorithms employ a depth-first memory layout to store tree nodes, whereas the distance heuristic uses a breadth-first memory layout to store tree nodes. For illustrative purposes an example BVH containing four triangle primitives labeled 1-4 is presented in Figure 2. The stackless algorithm always visits the left child first because the traversal order is fixed. It is most cache-friendly to store the tree in depth-first order because any given node is immediately followed by its left child:

In the case that the ray intersects all four triangles, there will be zero cache misses because the memory layout of the tree exactly matches the order in which nodes are visited.

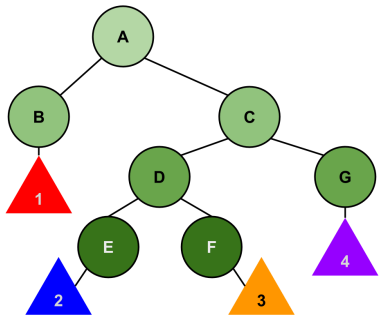

Fig. 2. Reference BVH tree

However, it is more likely that the ray fails to intersect a large proportion of nodes in the tree, so traversal will jump ahead in the linearized BVH to follow miss links. Miss links bypass entire subtrees, so these jumps will grow with tree depth. It is unlikely nodes at miss links will already be cached, resulting in sporadic cache misses and suboptimal memory usage. The same limitation applies to the split-axis heuristic algorithm, perhaps even more strongly because we expect the traversal algorithm to visit nodes out of order. Assuming one of the two children is chosen first at random, the right child will be visited first half the time, and cache misses would occur for half of all node traversals. However, the split-axis heuristic allows for front-to-back traversal, so we would expect cache misses to occur for less than half of all memory accesses in the linearized BVH.

The distance heuristic algorithm tests ray-box intersections with both children's bounding volumes at the same time to determine which should be pushed onto the stack first. This access pattern begets a breadth-first memory layout because data for both children must be read at the same time:

The breadth-first layout ensures that siblings are always adjacent in memory, so the right child will most likely already be cached once the left child is read, essentially resulting in a "free" memory access compared to an uncached read. However, child nodes are removed from their parent node by a distance that grows with tree depth. Since the capacities of caches on iGPUs are limited, we expect this benchmark to indicate which memory layout is

more cache-friendly, and additionally, how strongly cache utilization or underutilization will impact performance.

Additionally, grids will be constructed using CRH and the hierarchical structures will be constructed using SAH. Though the performance of actual tree or grid construction time is not of relevance in this study, construction methods have a strong influence on performance, and thus we expect usage of these state-of-the-art construction heuristics to help facilitate a fair comparison between all algorithms.

[3.2] Test Environment

All algorithms will be benchmarked on a laptop with an Intel Core i7-13700H processor, which supports Intel Iris Xe Graphics, Intel's standard integrated GPU architecture. All algorithms will be implemented using JavaScript and the WebGPU API. All source code for the algorithms and testing framework itself is provided via the following `remote repository`, under the branch name `wrtg3030_proj`. Since JavaScript is widely supported across web browsers and WebGPU abstracts over native APIs such as DirectX, Vulkan, OpenGL, and Metal, the testing framework is portable to a wide variety of devices, whereas prior tests have been hyper-specialized for state-of-the-art hardware. This suits the relatively larger set of devices that utilize integrated graphics hardware compared to a few mainstream dedicated GPU brands. However, time will not permit testing on more than one.

Each algorithm will be executed in a compute shader written in WebGPU Shading Language (WGSL), based on the pseudocode presented in Previous Work. Though recent research suggests a wavefront approach to be more performant than the traditional megakernel approach (Laine et al, 2013), we opt for a megakernel because Laine et al admit that the maintenance of ray pools and queues for their wavefront approach may be prohibitively expensive on low-bandwidth systems. Seeing as this study explicitly targets low-power, low-bandwidth hardware, the wavefront formulation does not seem well suited for our testing framework. Additionally, optimizing the distribution of rays to GPU threads is not the object of this study. As such, each ray is assigned a thread in workgroups of size $8 \times 8 \times 1$, effectively subdividing the image into 64-square-pixel tiles that will each be processed in parallel.

All images will be rendered in 720p (1280x720 resolution), constituting a total of ~1M primary or camera rays. We adopt Kajiya's rendering equation for our shading model and utilize a diffuse BRDF for all surfaces, so it is expected that rays will be scattered randomly after primary intersection, effectively creating ambient occlusion (AO) rays. With a uniformly distributed ray, the tests thus generalize over most existing BRDFs. This constitutes a total of ~2M rays or ~1M total paths computed per rendered frame. Ray workloads in the order of 10^6 appear to be fairly standard across most existing benchmarks, and indeed these workloads are reflective of real-world applications.

[3.3] Test Models

Three standard models have been selected for the benchmark, all of which have been provided for academic use by the labeled institutions:

Table 2. Selected 3D models for the benchmarks and accompanying triangle counts

1. *Teapot*, courtesy University of Utah
   2,256 triangles
2. *Bunny*, courtesy Stanford
   144,046 triangles
3. *Dragon*, courtesy Stanford
   871,306 triangles

Test models have been chosen such that all algorithms are tested on problem sizes of varying magnitudes. The selection is representative of real-world applications because scene size typically approaches 1M triangles. All models will be rendered from three distinct viewpoints in an attempt to limit the effect of choosing a vantage point from which any particular algorithm performs disproportionately well. This method seems to be typical in previous studies. Figure 3 presents the selected test models. A total of 64 frames will be timed from each model from each viewpoint, with 2 rays per pixel per frame. These measurements may or may not be of consecutive frames due to inherent synchronization issues between the execution of GPU timer queries and the mapping of results to a CPU buffer. This should not impact the results because timestamps for frames appear to be lost regularly, i.e. the probability that a timestamp is lost is constant or uniform. Thus the 64 frames can be seen as uniformly random samples from however many total frames needed to be rendered, say n total frames.

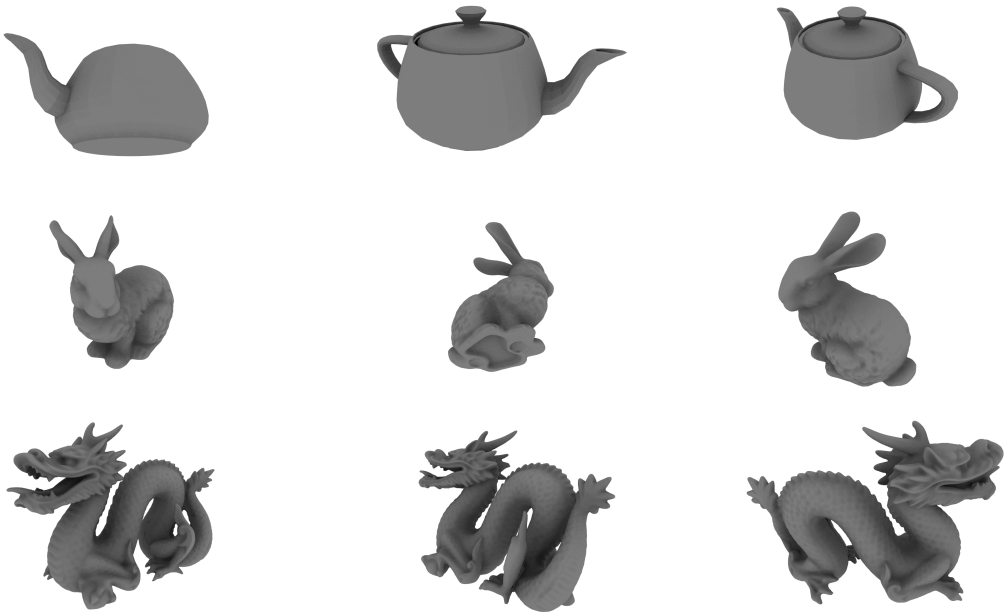

Fig. 3. Selected test models for the benchmark. The teapot *(top) has 2256 triangles, the* bunny *(middle) has 144K triangles, and the* dragon *(bottom) has 871K triangles. The models were chosen to reflect the variety in real-world problem size. Each of the three viewpoints used in the benchmark are also shown for each model.*

4. RESULTS & DISCUSSION

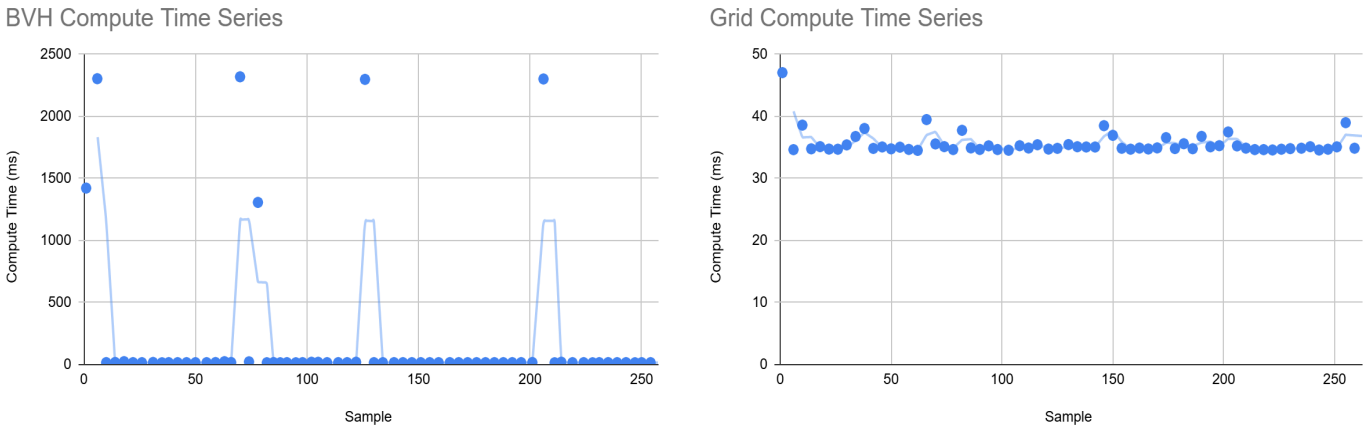

Fig. 4. Selected time series for render times for the same model and viewpoint. Note outliers in BVH (distance heuristic) (left) and comparatively consistent render times of Uniform Grid (right)

Initially, the plan to rank each algorithm was to take the mean of render times for each algorithm across all models and viewpoints. After the data was obtained this ranking methodology proved insufficient because different use cases may have different priorities. For instance, consider Figure 4, in which two specific time series for render times have been graphed: one for the distance heuristic-based BVH and one for the Uniform Grid. Notably, the distance heuristic time series is characterized by regular outliers, and this is common across all types of BVH and kd-tree. This is significant because both paradigms are hierarchical, suggesting a potential limitation of hierarchical models on integrated platforms. However, excluding outliers, the distance heuristic takes an average of ~15ms per rendered frame, compared to an average of ~35ms for the uniform grid. Dealing with the outliers thus proves problematic because it appears that BVH has the capacity to outperform a uniform grid, though a ranking based on a raw mean would cause BVH

to rank much lower than a grid because the outliers tank its averages.

The ranking scheme we thus adopt uses a weighted sum of a *truncated* mean and the standard deviation of render times. For each algorithm, this weighted sum is computed for each model and viewpoint and ultimately averaged across all test cases. The truncated mean averages the middle 90% of render times, thereby ignoring the remaining 10% of cases where an algorithm appears to perform abnormally better or worse compared to how it usually performs. Of course, this ratio can be adjusted as needed for different use cases. Yet the outliers in Figure 4, for instance, represent a render time that is over 100x slower than the truncated mean. From the perspective of the user, this represents a drop from 60 frames per second (fps) to over 2 seconds per frame, and this manifests as an abrupt and sporadic "stuttering" of the application that greatly hinders interactivity. Depending on the context, interactivity may be prioritized differently: in a clustered computing context, it may be less relevant; but for a game, for instance, interactivity and smooth framerate is prioritized above all else. Thus the standard deviation can be used to indirectly measure the "consistency" of framerate, and a weighted sum of these two metrics thus serves as a score with which all algorithms can be ranked. Weights can be adjusted depending on whether the application is meant to be real-time or offline. For this benchmark we use 0.5 for both weights, effectively averaging both metrics and lending equal importance to both of them. Table 3 ranks all five algorithms tested in this benchmark.

Algorithm	Name	Acceleration Paradigm	Construction Heuristic	Time Complexity	Space Complexity	Indexed Ranking	Normalized Ranking
3DDDA	`BVH_FTB_DF`	Uniform Grid	CRH	$O(\sqrt[3]{n})$	O(1)	1	1.000
Distance heuristic	`BVH_FTB_BF`	BVH	SAH	$O(\log n)$	$O(\log n)$	2	4.148
split-axis heuristic	`BVH_FTB_DF`	BVH	SAH	$O(\log n)$	$O(\log n)$	3	5.756
Stackless	`BVH_STACKLESS_DF`	BVH	SAH	$O(\log n)$	O(1)	4	6.234
kd-restart	`BVH_FTB_DF`	kd-tree	SAH	$O(\log n)$	O(1)	5	10.383

Table 3. Summary of algorithm characteristics and final ranking. The indexed ranking provides a raw ranking whereas the normalized ranking measures how much worse a given algorithm performed compared to the fastest, based on our modifiable ranking scheme. For instance, the distance heuristic was deemed 4.148x slower on average than 3DDDA, the fastest algorithm.

Reporting the normalized scores for each algorithm in addition to their raw indexed ranks is valuable because it provides an idea of approximately how much slower any given algorithm will perform than the best algorithm. The Uniform Grid performed the best by far on the integrated device tested, which contradicts the existing consensus on dedicated graphics cards.

As for the cause of the outliers themselves, several potential factors can be identified. The first is poor cache coherency: the memory access pattern of all BVH algorithms is sporadic and can even be modeled as random, as discussed in Materials & Methods. The lack of locality between traversed nodes in memory can feasibly result in frequent cache misses. Clearing and refilling the cache is not necessarily a cheap operation for the iGPU to carry out, especially given its relatively limited cache, so it is expected that any lack in cache coherence will have a more pronounced effect than on dedicated cards, which have a comparatively abundant pool of cache memory whose high bandwidth can hide poor utilization of the cache. Conversely, the Uniform Grid exhibits a highly predictable memory access pattern. The 3DDDA algorithm accesses grid cells in a fixed, repeating order. It is also likely that traversed cells will be adjacent in

memory, though the tabular memory layout of grid cells themselves means that the traversal algorithm may need to repeatedly access cells across separate memory pages, which constitutes poor cache utilization. Regardless, it is also significant that individual grid cells have $\frac{1}{4}$ the memory footprint of BVH tree nodes. Tree nodes require 24 bytes to store AABB minima and maxima plus an additional 8 bytes for pointers. Grid cells require only 8 bytes to store primitive list references, and traversal of the auxiliary triangle ID list is sequential and thus cache-friendly. Consequently, it is expected that 4x as many grid cells can be stored in the cache at any given time. These characteristics all suggest that the grid typically exhibits better cache coherency than the other algorithms, as the limitations of the BVH are applicable to the kd-tree.

However, cache underutilization is not typically expected to manifest in the form of performance spikes but rather by raising the average time across all rendered frames. It is more likely the hierarchical algorithms are more computationally-intensive, and integrated platforms typically respond to high energy usage by *throttling* the iGPU, or intentionally capping its clock speed until power and temperature metrics return to acceptable limits. This rationalizes the abrupt and long slowdowns, as the system is deliberately being stalled by design. It is expected that throttling may be a symptom of poor cache utilization because uncached reads are typically expensive.

The 3DDDA grid traversal algorithm is also constant-space, suggesting a seemingly obvious advantage over the two stack-based BVH algorithms. As discussed in [3], it is expected that constant-space complexity facilitates higher thread occupation and therefore better utilization of the iGPU's compute capacity at any given time. The relative speed of 3DDDA seems to confirm this, though it should be noted that constant-space complexity is not an immediate predictor of performance. The stackless BVH and kd-tree algorithms performed the worst despite being constant-space. The kd-tree, or at least the particular implementation used in this benchmark, seems to suffer from the worst aspects of both the BVH and the Uniform Grid. The poor cache utilization that appears to be characteristic of hierarchical models compounds with the costs incurred by maintaining an auxiliary array of primitive lists, ultimately tanking its performance. We additionally hypothesize that the lightweight nature of the 3DDDA traversal algorithm and its relative lack of floating point operations also bolsters its performance.

As for the stackless BVH algorithm, Figure 6 presents intersection heatmaps that offer unique insight into the behavior of each algorithm. Regions of "warmer" pixels indicate areas where more triangle intersections were computed. For this benchmark, ray-box intersections are treated as half as costly as ray-triangle intersections. Qualitatively, the work done in each algorithm appears relatively uniformly distributed except for the stackless algorithm (bottom right). Evidently, the fixed traversal order of the stackless algorithm resulted in suboptimal thread utilization, with some threads searching deeper into the tree than others. Typically, threads in a workgroup will execute in lockstep, i.e. all threads execute the same instruction at the same time to leverage the highly parallel SIMD and SIMT microarchitectures on GPUs. Consequently, should the control flow of the compute shader diverge and any threads perform a disproportionate amount of work compared to their sibling threads, the workgroup will effectively idle and "stall". As such the stackless traversal scheme exhibits both suboptimal cache and thread coherence, illustrating the importance of benchmarking from multiple viewpoints because the stackless scheme clearly performs more favorably at particular viewpoints where ray direction and split axes align (Figure 5).

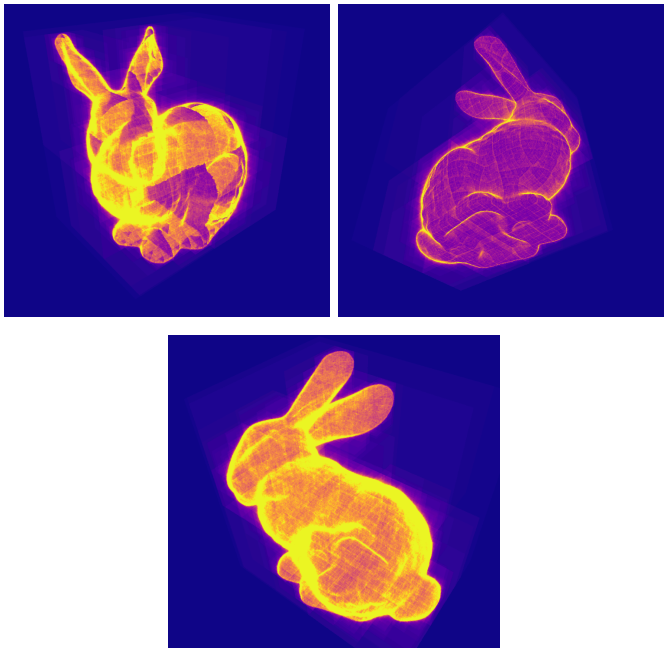

Fig. 5. Stackless BVH heatmaps illustrate how fixed traversal order results in poor thread occupation.

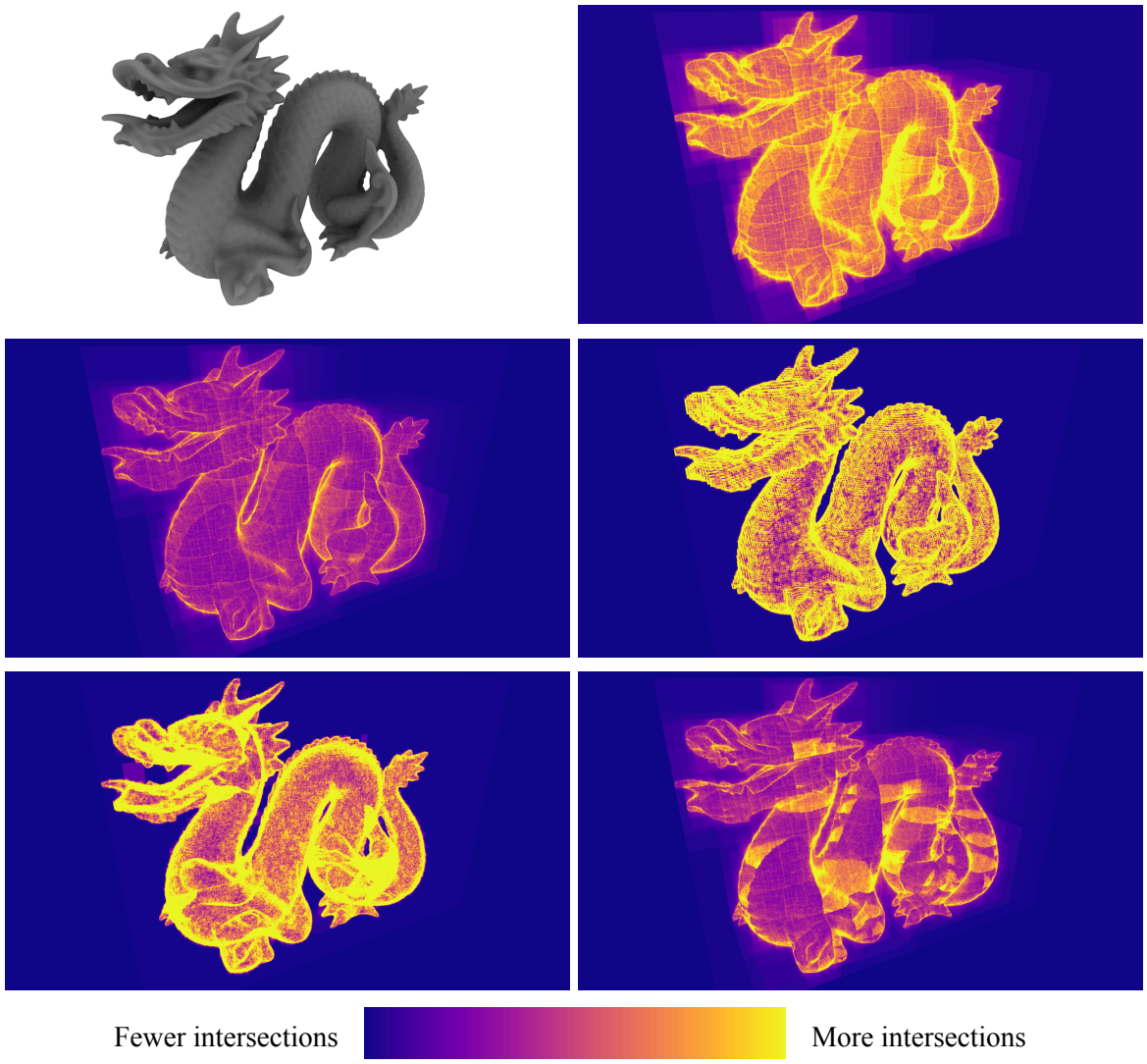

Fig. 6. Heat maps for dragon*. "Warmer" colors indicate pixels where more triangle intersections have taken place. Going top down, left to right, the images are: viewpoint reference, BVH (distance heuristic), BVH (split-axis heuristic), Uniform Grid, kd-tree, & BVH (stackless)*

Figure 5 is particularly interesting because the two fastest algorithms do not appear to minimize ray intersections. Qualitatively, the split-axis heuristic BVH minimizes the number of ray intersections conducted most effectively, yet it ranks 3rd and nearly 6x as slow as the Uniform Grid. Firstly, this demonstrates that the breadth-first memory layout of the distance heuristic BVH is more cache-friendly than the depth-first layout employed by the split-axis heuristic BVH–if the distance heuristic BVH did not benefit from its cache utilization, then it would have performed more poorly than the split-axis heuristic on account of its extraneous intersections. More generally, it seems that memory and cache usage demonstrate the most profound effect on acceleration paradigm performance. Consider Figure 7, which contrasts the measured and theoretical runtimes of all algorithms. The Uniform Grid's cube root complexity is theoretically inferior to the logarithmic runtimes of all other algorithms, and theory dictates that it should not have scaled as well as it did, particularly in comparison with the others. Ultimately, this suggests that on limited platforms such as integrated GPUs, hardware factors are of much more importance than on other systems. Indeed, dedicated graphics cards are closer (though not necessarily close) to the ideal random-access machines that theoretical evaluations like big-O are based upon. Consequently, as the benchmarks have shown and Figure 7 emphasizes, existing theoretical metrics for algorithm design are no longer applicable to integrated devices. If they were, the logarithmic-time, constant-space algorithms would have

performed the best, yet here they were the slowest. The assumption underlying the design of acceleration paradigms does not appear to hold on integrated platforms: minimizing ray intersections does not always optimize performance. As such, we suggest that new design priorities be adopted for the future development of acceleration paradigms on iGPUs. Particularly, the results imply that given intersection time is already sub-linear, cache coherence, thread occupation, and energy usage exceed all other considerations. Cache and thread utilization are relevant on other platforms but perhaps not to the same degree as they are on integrated devices, particularly because they are already computationally constrained. Together, these factors appear to impact energy usage, an especially novel factor that has a profound effect on performance. Should energy usage grow too high, it has been demonstrated that power and temperature throttling can occur, which has negative implications for average render time. Maximizing the use of limited hardware given energy constraints can thus serve as a mantra for future algorithm design. For the time being, it appears that the Uniform Grid is most suitable for integrated applications.

Future work could include compression of hierarchical models to improve cache locality, and perhaps more importantly, a more extensive study on the energy consumption of different algorithms and the factors that impact it outside of the two identified in this paper.

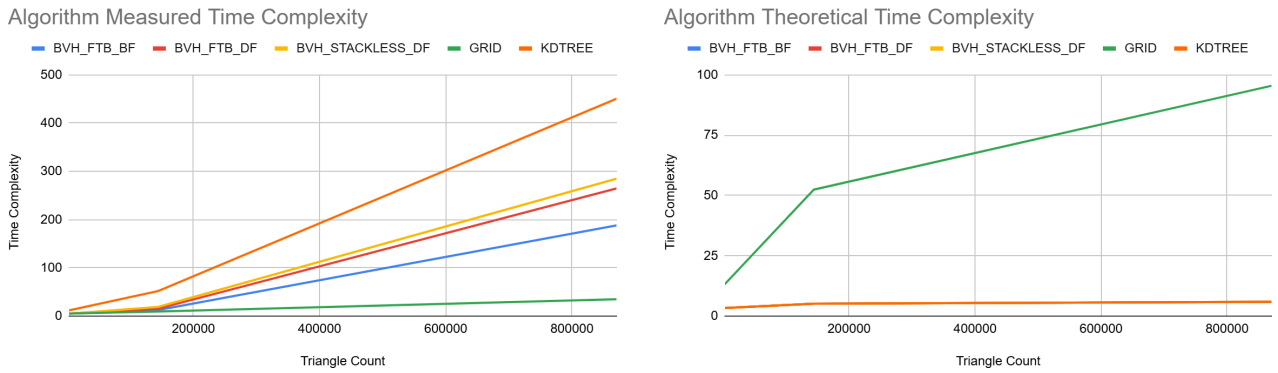

Fig. 7. Theoretical (right) vs. actual (left) runtime complexity, based on the size of test scenes (measured in triangles). Note the discrepancy between the projected runtime of the Uniform Grid and its actual runtime.

REFERENCES

01. Arthur Appel. 1968. Some techniques for shading machine renderings of solids. In Proceedings of the April 30--May 2, 1968, spring joint computer conference (AFIPS '68 (Spring)). Association for Computing Machinery, New York, NY, USA, 37–45. https://doi.org/10.1145/1468075.1468082
02. James T. Kajiya. 1986. The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86). Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/15922.15902
03. Timothy L. Kay and James T. Kajiya. 1986. Ray tracing complex scenes. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86). Association for Computing Machinery, New York, NY, USA, 269–278. https://doi.org/10.1145/15922.15916
04. James H. Clark. 1976. Hierarchical geometric models for visible surface algorithms. Commun. ACM 19, 10 (Oct. 1976), 547–554. https://doi.org/10.1145/360349.360354
05. A. Fujimoto, T. Tanaka and K. Iwata, "ARTS: Accelerated Ray-Tracing System," in IEEE Computer Graphics and Applications, vol. 6, no. 4, pp. 16-26, April 1986, doi: 10.1109/MCG.1986.276715.
06. Marek Vinkler, Vlastimil Havran, and Jiří Bittner. 2014. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In Proceedings of the 30th Spring Conference on Computer Graphics (SCCG '14). Association for Computing Machinery, New York, NY, USA, 29–36. https://doi.org/10.1145/2643188.2643196

07. Daniel Meister and Jiří Bittner, Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing, Journal of Computer Graphics Techniques (JCGT), vol. 11, no. 4, 1-19, 2022.
08. Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In Proceedings of the Conference on High Performance Graphics 2009 (HPG '09). Association for Computing Machinery, New York, NY, USA, 145–149. https://doi.org/10.1145/1572769.1572792
09. Dominik Wodniok, Michael Goesele, Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees, Computers & Graphics, Volume 62, 2017, Pages 41-52, ISSN 0097-8493, https://doi.org/10.1016/j.cag.2016.12.003.
10. Seo, W., Park, S., & Ihm, I. (2022). Efficient Ray Tracing of Large 3D Scenes for Mobile Distributed Computing Environments. Sensors (Basel, Switzerland), 22(2), 491. https://doi.org/10.3390/s22020491
11. P. Andrade, T. Sabino and E. Clua, "Towards a heuristic based real time hybrid rendering a strategy to improve real time rendering quality using heuristics and ray tracing," in 2014 International Conference on Computer Vision Theory and Applications (VISAPP), Lisbon, Portugal, 2014, pp. 12-21
12. J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," in IEEE Computer Graphics and Applications, vol. 7, no. 5, pp. 14-20, May 1987, doi: 10.1109/MCG.1987.276983.
13. Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In Proceedings of the Nineteenth Eurographics conference on Rendering (EGSR '08). Eurographics Association, Goslar, DEU, 1225–1233. https://doi.org/10.1111/j.1467-8659.2008.01261.x
14. Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic ray stream traversal. ACM Trans. Graph. 33, 4, Article 151 (July 2014), 9 pages. https://doi.org/10.1145/2601097.2601222
15. Brian Smits. 1998. Efficiency issues for ray tracing. J. Graph. Tools 3, 2 (1 February 1998), 1–14. https://doi.org/10.1080/10867651.1998.10487488
16. Tim Foley and Jeremy Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS '05). Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/1071866.1071869
17. J. G. Cleary, B. Wyvill, R. Vatti, and G. M. Birthwistle. 1983. Design and Analysis of a Parallel Ray Tracing Computer. In Proceedings of Graphics Interface ’83 (GI ’83), Edmonton, Alberta, Canada.
18. Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. 2006. Ray tracing animated scenes using coherent grid traversal. ACM Trans. Graph. 25, 3 (July 2006), 485–493. https://doi.org/10.1145/1141911.1141913
19. Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In Proceedings of the 5th High-Performance Graphics Conference (HPG '13). Association for Computing Machinery, New York, NY, USA, 137–143. https://doi.org/10.1145/2492045.2492060