Batching of divergent rays on GPU architectures

Nick Begg

Supervisors: dr. Ing. J. Bikker, dr. A. Vaxman

Utrecht University, Faculty of Science, Department of Computer Science

ICA-5968577

Abstract

Ray tracing is an image generation method that involves fundamentally incoherent access to system memory. Ray batching is a technique developed to defer certain accesses to memory when ray tracing, to allow many similar accesses to be performed together - increasing the effectiveness of modern processors' caches. This research aims to take an existing successful ray batching system originally designed as a single-threaded CPU based system, and adapt it for use in a GPU environment. A technique is presented which successfully performs batching on a GPU, however no performance improvement is demonstrated over current implementations. Insights into the performance of varying memory layouts and data structures, and parallel access required to access these are gained.

Universiteit Utrecht

Contents

1. Introduction **4**
1.1. Background 4

2. Related Work **6**
2.1. Packet Tracing 6
2.2. Ray Batching 6
2.3. Hardware Based Implementations of Ray Batching 7
2.4. Traversal Algorithms 9
2.5. Reduced Precision and Compressed Acceleration Structures 12
2.6. GPU architectures 12

3. Implementation **15**
3.1. Overview 15
3.2. BVH Generation 15
3.2.1. Main work loop 16
3.2.2. New Ray Generation and Next Work 17
3.2.3. Next Bucket Selection 17
3.3. BVH Traversal 18
3.3.1. Top-Level Traversal Overview 18
3.3.2. Top-Level Traversal Parallelism and Algorithm Variations 19
3.3.3. Treelet Traversal 20
3.4. Rays 20
3.4.1. Ray Storage 20
3.4.2. RayQueues Overview 20
3.4.3. Linking and Delinking RayQueues 21
3.4.4. Enqueuing and Dequeuing 21
3.5. Locking and Parallelism 22
3.6. Occlusion v. Closest hit 22
3.7. Shading 23
3.8. Random Number Generation and Seeding 23
3.9. Debugging Tools 23

4. Experiments and Results **24**
4.1. Overview 24
4.2. CUDA Profiling 24
4.3. Initial Run and Tree-Size Testing 25
4.4. Ray Count Experiments 26
4.5. Queue Size Experiments 26

5. Discussion **29**

6. Conclusions And Future Work **30**
6.1. Parallelism and Improved Traversal Strategies 30
6.2. More Granular Measurements 31
6.3. Multi-tier Trees 31
6.4. Multi-queuing Rays and Fairer Queuing 31

6.5. Dynamic Allocation 32
6.6. Memory Access 32

7. Acknowledgements **33**

8. References **34**

A. Structure Listings **39**

B. Tuning Parameters **40**

C. Initial-Run Results Table **42**

D. RayCount-Run Results Table **46**

E. QueueCount-Run Results Table **48**

1. Introduction

The ray tracing family of algorithms are methods of image generation finding increasing application within real time graphics. Whilst fairly well established within offline image generation environments - such as image rendering for movies and television - it is now becoming feasible to apply these methods to real-time image generation. This is due to, in part, the significant increase in compute power available in recent GPU implementations, as well as developments in dedicated hardware for processing parts of the problem.

Ray batching (also known as ray scheduling) is an optimisation for one particular challenge of ray tracing; that is that memory access when ray tracing is effectively random, leading to sub-optimal performance. Ray batching attempts to make access to memory more coherent by forming ad-hoc or transient groups of rays as they progress through the ray tracer's data structures. In particular, we will take the work of Gasparian and Bikker [Gas16], a ray batching system designed to run in a single-threaded CPU environment, and research its applicability to a GPU environment.

1.1. Background

Practically any modern computer system used for ray-tracing will have a multi-tier memory hierarchy, with each level having a specific storage capacity, locality (such as per-core or global) and performance characteristic. For example, in a typical contemporary desktop computer system with an Intel *Kaby Lake* CPU, there will be a 3-tier CPU memory cache, with the L1 cache (closest to the CPU core) 64KB in size, L2 256KB and L3 around 2MB. Main system RAM is typically sized in the order of gigabytes. The L1 and L2 cache will exist per CPU core, and the L3 will be shared across all cores. In addition, the system will have hard drive or SSD storage typically sized in the terabytes; this can also be considered another memory tier. Even access to the main system ram can be optimised - DRAM chips themselves contain a buffer in front of the main storage. Optimising access by being aware of charge cycles and recent access can improve performance [Has+18].

By its very nature, ray tracing requires random access to memory for both intersection and occlusion queries. This random access also poses a problem for caching, quickly making caches ineffective for a naïve implementation. For the lifetime of rendering a single frame, much of the memory access is read-only, making the algorithms very easily parallelisable. As processing power has rapidly increased, has caused ray tracing performance to become dominated by memory bandwidth limits rather than computational capacity[Wal+01].

This has lead to a multitude of techniques to improve ray tracing performance, most of which boil down to (at least in part) carefully controlling which parts of the scene geometry are accessed, and in what order. In particular a focus has been on doing more work with the data fetched from RAM - effectively getting better 'value for money'.

A series of data structures have been devised to divide the 3D space and optimise geometry queries - both in terms of compute resources and memory bandwidth usage. These include manually constructed bounding structures [RW80], voxel grids [AW87], hierarchical spatial subdivision [Gla84], BSP trees [SS92], K-D trees [Hav00] and finally, the bounding volume hierarchy (BVH).

The acceleration structure aims to solve one side of the problem - that is, to subdivide and group subsets of a scene's geometry, and provide a quick traversal path to a required subset. The other side of the problem is to try to group rays together based on common paths through the scene, and allow them to traverse an acceleration structure together, thus allowing the cost of fetching and processing acceleration structure nodes and geometry to be amortised across many rays.

There are a number of strategies for grouping rays. A number of advanced traversal algorithms have been devised in recent years that increase ray coherence by bundling active rays together dynamically as they traverse an acceleration structure, such as Tsakok [Tsa09], DRST [BA14] and ORST[Fue+15].

Ray batching aims to create transient ray queues throughout points on an acceleration structure as rays traverse by allowing the traversal algorithm to deposit rays in queues, attached to points throughout the structure. Once a given queue has filed, the portion of the acceleration structure to which it is attached is traversed by the queued rays. Gasparian and Bikker's system[Gas16], informally known as *RayCrawler* is the starting point for this research, and can be considered a state of the art implementation of a ray batching system and advanced traversal implementation, comparable to ORST.

2. Related Work

2.1. Packet Tracing

One alternative method to ray batching intended to reduce memory incoherence is so-called *packet* or *pyramid traversal*. A packet of rays is launched in a frustum from the camera point into the scene, and the intent is that some degree of coherence will be observed in the set of geometry intersected. Exploiting this coherence will amortise the cost of memory access over the entire packet. Various notable examples of this method include van der Zwaan et al[vdZRJ95], Wald et al[Wal+01], Reshetov et al[Res05], Boulous et al[Bou+07] and Garanzha et al[GL10].

Considering diminishing coherence for divergent ray distributions in path tracing [Kaj86], several authors suggest using wide BVHs instead of packet traversal. In particular, cpu vector instruction sets such as AltiVec, SSE, AVX and AVX-512 benefit from 4-, 8- and 16-wide BVH traversal[DHK08][EG08][WBB08].

Benthin et al.[Ben+12] devised a hybrid traversal model that combines ray packets and single ray traversal.

2.2. Ray Batching

Pharr et al sought to address the problem as a scheduling challenge [Pha+97]. They developed a system designed to handle scenes far too large for system memory, based on voxels. When tracing through a given voxel, required data would need to be explicitly fetched from disc before tracing could continue into that segment. They also ran into issues with the sheer volume of rays that are spawned by the tree structure of by Whitted-style ray tracing [Whi79], especially when they scheduled a large number of active concurrent traces rather than the traditional depth-first implementation. Their scheduling algorithm was based around a two-tier storage model - that is system memory, and disc. As access to disc was orders of magnitude slower, system memory was largely used as a "cache" for state that had been fetched from disc, optimising access to these memory-based "caches" was a primary focus. The CPU's multi-tiered memory caches themselves were largely ignored during optimisation.

Navrátil et al[Nav+07] develop these ideas further, considering Whitted's algorithm and Pharr et al's as two points on a continuum of the number of simultaneously active rays a system can handle - and attempt to find an optimal point on this line. Unlike Pharr et al's 2-tier memory/disc model, Navrátil et al optimised their system around the L2 cache of a contemporary CPU (though it could be generalised to arbitrary cache models and sizes), using a ray representation that could be packed into 64 bytes. They also looked to limit the explosion of ray numbers inherent to the recursive nature of Whitted-style [Whi79] ray tracing, which was an issue for Pharr et al. This was addressed by spawning secondary rays in generations.

As they could determine the exact storage requirements of the cumulative geometry and acceleration nodes below any given point in their acceleration structure, their queue point selection algorithm would select the set of nodes closest to the root in the tree such that each queue point's children would fit entirely in L2 cache; indeed, if the whole tree fit in L2 cache, the algorithm regresses to regular K-D

tree traversal. Starting with regular top-down traversal, as rays reached a node with a queue point, they would be placed in the given queue for later continuation. When a queue filled, all rays in the queue would traverse the given sub-tree, including any leaf nodes. Thus the cost of fetching the subtree and geometry was amortised across the whole ray queue. They also implemented a special case for leaf-nodes that exceed L2 cache size, which would normally result in thrashing the cache for ray every traversal - this was achieved by iterating all rays over cache-sized chunks of geometry.

Ultimately they did not propose a new traversal algorithm per se, but provided a ray scheduling method that could transparently queue and restart rays as required for improved cache performance. This research resulted in an improvement in memory traffic requirements, but the results were only simulated.

Eisenacher et al [Eis+13] devised a CPU-based, two-level quad-BVH system with very large ray batches to handle scenes much too large for system memory. A four stage pipeline processes each iteration of a ray's life. At the start of the pipeline, the large set of active rays (typically 30-60M) are sorted into bins based on direction, and stored in a compressed format (36 bytes per ray, using lossless octahedral normals [Mey+10] for ray directions). Once a bin reaches a fixed size, the batch is streamed to disc. As traversal threads become idle, ray batches are read back in from disc, and traversed across the main-BVH (using streaming packet traversal [GL10]) and child-BVHes (using naïve traversal). The directional coherence of the sorted rays gives and approximate front-to-back traversal ordering. As the rays intersect with geometry they are binned together, based on the particular intersection mesh. Finally, shading is performed in mesh order, meaning that the cost of retrieving shading data from disc is spread across all intersecting rays.

A mobile-focussed MIMD hardware design was presented by Lee et al [Lee+15] which aims to passively reduce memory accesses by actively reordering rays based on existing cache status. Its ray traversal hardware uses a design called "reorder buffer", which implements a batching BVH traversal method, inspired by the out-of-order non-blocking cache [FJ94]. It treats individual rays as a kind of thread, which can be scheduled for processing by the hardware; In particular the cache status of the acceleration structure or geometry data required by each ray is used to either queue or activate individual rays, much like the latency hiding methods of programmable processors. Lines in the cache are tagged to rays which depend upon them; This way, as a cache line becomes populated, dependent rays can be scheduled together, ultimately reordering the traversal processing to match the cache status. As the traversal hardware is a fixed function design, and works at the level of individual rays, it does not suffer from potential occupancy issues that SIMT designs experience.

Bikker [Bik12] queues rays in the leaves of a shallow octree and traverse mini-BVHs for these leaves, achieving improved performance for in-core rendering compared to single ray traversal.

2.3. Hardware Based Implementations of Ray Batching

Many attempts have been made to implement some or all parts of the ray-tracing pipeline in custom hardware - In most cases, the successful projects were implemented on FPGA hardware. Ultimately these designs faced similar challenges to processor based solutions - namely that the random-access nature of ray-tracing generated a large amount of memory traffic, and that optimising this traffic became a focus to improve performance. In addition, the nature of shading requirements imposed a need for processor-like instruction sets.

Because hardware acceleration structure and geometry traversal units were typically implemented as fixed-function designs, independent of acceleration structure generation/refit, ray generation and shading units, in effect most of these systems involved some degree of ray batching or queueing as rays entered or left the various components.

D-RPU included a hybrid BVH and KD-tree referred to as a bounded-KD (B-KD) tree [WMS06]. As well as being able to traverse this structure, the design could also re-fit an existing tree.

SGRT implemented a dual-AABB based BVH[Lee+14]. Viitanen et al expanded upon this [Vii+16] to build a hardware implementation of an MBVH [EG08].

SaarCOR [SWS02] uses large ray packets, and implemented fixed-function hardware for acceleration structure traversal, intersection and shading. Further research [SLa03] proposed a dedicated virtual memory system for the accelerator; This system would use a local DRAM as a cache for the accelerator, independent of a main system RAM. This virtual memory system was never built; Problems with the design were identified when large changes to a scene's working set occurred. Pre-fetching was considered as a solution, but this was not explored further [Woo06].

SaarCOR was subsequently expanded into RPU [WSS05] which added a programmable intersection unit (allowing for higher-order primitives to be intersected) and programmable shading. Finally D-RPU [Woo06] built upon RPU, which added a B-KD tree acceleration structure [WMS06], and included on-chip refitting of this structure. In addition, programmable shading was improved, with a GPU-like processor allowing for improved shading programs, including the ability for shaders to instigate their own recursive ray traces.

TRaX [Spj+09] is a programmable accelerator design specialised for ray-tracing, including a host simulator and an LLVM[LA02] based compiler. TRaX used a multithreaded MIMD design to optimise handling of divergent ray paths. TRaX was further enhanced [Kop+10] with increased shared caches and functional units. STRaTA [Kop+13] expands TRaX with specialised pipelines – for BVH intersection and triangle intersection. These pipelines are controlled through the programmable interface using special instructions and ultimately implements a treelet based ray batching system optimised around the size of the L1 cache.

Aila and Kerras [AK10] proposed a GPU-like architecture for treelet-based subdivision of an acceleration structure, to allow for dynamic ray scheduling to manage memory bandwidth consumption. Their design consisted of multiple proposed models of operation, starting with a stack memory layout of Aila and Laine [AL09]. Strikingly, they determined that stack traffic is responsible for approximately half of the total traffic.

Their initial finding was that parallel traversal of independent rays results in a very large working set. To alleviate this, they build upon Navrátil et al's [Nav+07] model with a treelet-based BVH subdivision, queuing rays at the top of the treelets. They also point out that Navrátil's design assumed that all data from their root BVH stayed in cache, and did not measure the traffic between relating to this. Furthermore, when a ray exited a treelet, it re-entered traversal from the root of the tree.

They offer an extended model where treelets may contain other treelets; That is, a *cut* may be placed between any two nodes, and the child becomes the root of a treelet; A ray queue is also placed at the root of the treelet. The size of each treelet was optimised based on two factors - trying to find the treelets which balanced between having the largest collective surface area (in the spirit of the Surface Area Heuristic [MB90]) and those which would approximately match the size of the system's L1 cache. Also, they attempted to reduce the total number of treelet transitions, as this resulted in increasing queue memory traffic. They then found that stack traffic constitutes 17–28% of overall memory communication when treelets are used.

Scheduling methods are also presented as a vector for experimentation, with 2 presented; A Lazy scheduler (when idle, binds the current processor to the queue with the most rays), and a Balanced scheduler (tries to keep all queues at a given target size). These algorithms' role is to assign ray queues to processor resources. They showed that one algorithm was not implicitly better; rather that memory traffic was reduced for certain sized treelets depending on the scheduler chosen.

Interestingly, whilst traversal stack traffic was shown to be significant in memory traffic generation, they did not experiment with other traversal methods (in particular, stackless methods) or wider trees, instead listing these as an option for future work.

Nah et al [Nah+11] developed a hardware based traversal and intersection engine, built around Whitted-style [Whi79] ray tracing. This ultimately lead to the RayCore [Nah+14] system - a mobile focussed pure solution with K-D tree traversal and construction. Rays are processed through the system in a 4 stage macro pipeline (ray setup, T&I, hitpoint calculation, shading), with one progression through this pipeline representing a single bounce of a ray. At the end of the pipeline, rays can queue to come around again, or be terminated as required.

The core of the traversal mechanism is the so-called *T&I* (traversal and intersection) unit, which has its own internal pipeline. The unit contains four identical pipelines, each being of a unified design, able to process its currently active ray through traversal and intersection; At any given time, each pipeline operates in one of three modes - ray/box intersection, traversal or ray/triangle intersection (that is, leaf processing).

Like other fixed-function hardware based solutions, the ray is treated somewhat like a thread in programmable designs, and certain latency-hiding concepts are implemented at the ray level. Each pipeline in the T&I unit contains its own dedicated L1 cache, and the four pipelines of the T&I unit share an L2 cache. When an L1 miss occurs, the ray is marked as idle and continues through the 20 cycle pipeline (rather than stalling the pipeline). When the idle ray reaches the end of the pipeline, it is returned to the start of the pipeline to be retried. The same process is repeated on an L2 miss, although this time, the request goes to main memory.

Keely[Kee14] presents a hardware design that batches rays based on treelets, and allows immediate traversal - that is to say, proceeding past a batching point without queueing - in the case when required data is already in cache. Mathematical precision is reduced and BVH compression is employed to reduce circuit complexity.

Most of the aforementioned projects are implemented as largely self-contained systems; That is, they have moved most or all of the process of ray-tracing to their custom-built solution, and the CPU takes up a supervisor or driver role. Little-to-no intra-frame communication takes place, and instead the CPU transmits per-frame data to the accelerator in bulk for processing. Departing from this model, SGRT [Lee+12] is mobile-focussed hybrid design, re-using an existing GPU for shading and ray generation, a CPU for acceleration structure creation and custom hardware for acceleration structure traversal and intersection calculation, as well as refitting existing BVHes.

NVIDIA's Turing architecture[NVIa, p. 30-31] contains some elements of ray batching - or at least wavefront tracing. With its dedicated bounding box and ray-triangle intersection hardware, rays are scheduled between the SM units and dedicated BVH traversal and intersection hardware.

2.4. Traversal Algorithms

Naïve traversal of a BVH on a CPU is a fairly straightforward affair; frequently it is implemented as a recursive descent through the tree.

Many methods have been developed that either devise their own stack (rather than the implicit one provided by the compiler), or use a non-stack based method to track their state. Whilst these give a performance boost on a CPU, they are critical on a GPU as recursion is frequently not available at all. Furthermore, a method independent of the compiler's generated call stack is required in ray batching as the lifetime of a ray is now completely decoupled from an individual thread's walk through the BVH tree.

Some methods are categorised as *stackless* - these generally involve the use of parent pointers to walk upwards through the tree, and/or some kind of stack-like LIFO structure, often tightly packed into a bitfield.

Hughes et al devised Kd-Jump [HI09] a traversal method for ray tracing on GPUs using K-D trees, which is also possibly applicable to a BVH. The method is built upon Wald et al's implicit K-D tree [Wal+05]. In particular, it leverages the balanced nature of these trees to simplify the mapping between a node's logical position in the tree, to its array index. Coupled with a fixed lookup table generated a tree construction time (and stored in the GPU's constant memory), jumping up multiple levels of the tree in constant time.

Traversal state is then tracked using a bitfield stack. That is, the next required branching point is recorded using a bitfield. Instead of iteratively walking back up the tree, the number of levels to *jump* up is determined using the bitfield, then this jump is executed in one operation.

Laine et al developed a limited stack-based method that would restart when insufficient storage space is available to store required state, at the cost of having to re-traverse parts of the BVH[Lai10].

Hapala et al proposed a stackless BVH traversal algorithm for ray tracing [Hap+11] based on an iterative method with parent pointers. This method shares some similarity with kd-Jump [HI09], importantly however it does not rely on the ability to statically map between a node's logical position in an acceleration structure tree, and an array index to it; Kd-Jump requires its trees to be balanced, giving a predicable mapping; Hapala's method does not, giving more flexibility in the layout of the tree. Whilst it does revisit nodes, it intersects each node only once.

Their method is built upon standard binary BVH traversal, and requires that a ray's traversal be deterministic - That is for a given ray and BVH, the traversal order should be repeatable. However, different rays need not traverse the tree in the same order. In addition, for a given node, both the parent and sibling node (for interior nodes) should be cheaply determinable; the exact method is an implementation detail. For example, parent pointers can be held externally to the tree, or stored within the node. Determining a sibling may be determined by walking up to the parent, then choosing the *other* child.

Ultimately the method determines the traversal order using a simple state machine -

- if traversed down from the parent, descend to the first child.
- If traversed up from the first child, traverse to the second.
- If traversed up from the second child, traverse up to the parent

This method could be extended to other structures, such as a BVH4, at the cost of a slightly more complex state machine.

Tsakok [Tsa09] devised a method using a BVH4 that allows for efficient traversal of coherent rays whilst still allowing fast single ray traversal when no coherency is present; however all rays must traverse the tree in the same order, resulting in extraneous node traversal. It combines the benefits of a wide style BVH such as a QBVH [DHK08] or multi-BVH (MBVH) structure [EG08] [WBB08] with stream ray tracing [GR08], using a method referred to as "Multi-BVH Ray Stream Tracing" (MBVH RS).

The nodes of the BVH tree store a 4-wide set of child bounding boxes in structure-of-arrays form, suitable for fast testing using SIMD intersection tests as described in [EG08]. Large groups of rays traverse the tree breadth first and are intersected against each BVH node together, amortising the cost of the node fetch against the whole ray group. As the rays are 'filtered' out during descent the SIMD hardware is kept occupied, as the number of bounding boxes per node is constant. During traversal

of leaf nodes, a temporary SIMD packet of rays is built from the active set, and tested against the geometry of that leaf. The performance ray stream tracing is highly dependent on the hardware's scatter and gather support [GR08].

Tsakok's method keeps the SIMD occupancy of the processor high, as each ray is individually checked against an entire bounding box in one go. As the traversal descends, the set of rays in a stream stays constant. If a given ray is not interested in traversing a particular leg of the BVH, it is not processed during that phase. However, this means the memory cost of fetching the BVH node is shared across fewer rays.

Dynamic Ray Stream Traversal (DRST) [BA14] builds upon Tsakok's and similar methods to allow ray traversal with varying order, implemented for both BVH2 and BVH4 trees. Instead of keeping a constant stream of rays, it continuously rebuilds streams with each iteration of descent, using only rays that are interested in following a particular path. As large sets of rays are dynamically bundled together, SIMD intersection can be used to test multiple rays against a given bounding box at the same time. A shared traversal stack is maintained representing the active work remaining for the current set of rays. Additionally, it has a special case to fall back to single ray traversal in cases when few rays wish to take a given path.

The method also looks to optimise traversal based on intersecting nearest bounding boxes first - increasing the likelihood of early ray termination. For a BVH2, this is accomplished by generating up to 3 new ray streams at each iteration - streams 1 & 3 representing left, and stream 2 representing right. For rays that desire to go both left and right at a given node, if they want to go left first, they go into streams 1 & 2; Equivalently, if they want to go right first, they go into streams 2 & 3. The order of these streams is maintained as they enter the traversal stack, and if early termination occurs, a ray can be removed from the stack entirely.

A similar method is implemented for a BVH4, however it involves an approximation to consider it like a double BVH2, and to keep the number of potential stack allocations under control - at most 9 stack items will be allocated per iteration.

Ordered Ray Stream Traversal (ORST) [Fue+15] extends the concept of DRST specifically for BVH4 traversal using lookup tables to reduce the required bookkeeping when determining traversal order. In DRST, this bookkeeping involves a large amount of fragmentation, and potential for memory traffic. Unlike DRST, ORST allows traversal over child nodes in any order, as opposed to only 8 out of the possible 24.

The algorithm initially sorts rays by their direction signs - giving 8 bins of rays going in the same direction. These rays will then be processed together with an expected traversal order.

The heart of the traversal algorithm replaces DRST's stack reconstruction with a series of lookup-tables (LUTs) to define the traversal order. Nodes in the BVH4 now have a *perm* field added, which defines the topology of how the 4 child nodes are subdivided, and on which axes they are split. When traversing a node, by combining the perm field of the current node and the signs of the active ray, and index into the *orderLUT table* is generated. This leads to an order index, which combined with the an active mask (ie which child nodes of the current node are active), into the *compactLUT table*, which gives an ordered list of active child node indices.

Their method also includes BVH4 adaptations of Wald et al's [WBS07] coherent large packet traversal. The implementation combines conservative early miss, speculative early hit, ordered traversal, active ray tracking, and a packet test of last resort in a single unified traversal step.

Binder and Keller[BK16] developed a stackless method tuned for GPUs. Primarily their method involves using a 2 dimensional hash table for back-tracking to unvisited nodes; that is, for each node the hash table is used to determine the index of the n-th generation uncle - bypassing a 'parent pointer'.

A bitstack is used to determine the next postponed node. After implementing the hash table, they observed that 57% of hash lookups involved either the sibling, uncle or grand uncle. To optimise for this, the most recently postponed node's index is stored in a register(often the sibling), and the uncle and grand-uncle's indices are encoded within the node itself, saving a hash lookup. Finally they discard unreachable nodes - that is with a disjoint t-interval - after an intersection is found in the current node, to avoid pointless backtracking.

The authors reported between 8% and 20% average speedup over Aila's stack-based method [AL09]. It is not immediately clear how this could be adapted to BVH trees with wider branching factors.

Gasparian and Bikker's method [GB17] includes a novel multi-tier BVH4-based method, using a bitstack to record branching progress through the main BVH.

2.5. Reduced Precision and Compressed Acceleration Structures

A common and straightforward way to implement a BVH tree's node layout is to use 6 32 bit floating point variables for its bounding box (that is, 2 co-ordinates in 3D space) and 2 32 bit integers to reference its child nodes (for an interior node) or its triangle set (for a leaf node). This requires 32 bytes of storage in total, which has the favourable property of allowing 2 BVH nodes to fit in one standard CPU cache line.

However, as much of the memory traffic during raytracing is generated by fetching acceleration structure nodes, research into reducing the storage required per node has yielded positive results. Techniques such as reduced precision numerical values and using relative rather than absolute co-ordinates are commonly seen. Additionally, extra information is often required in a BVH node, which can be packed into a few spare bits. For example, ORST [Fue+15] requires *perm* and *index* parameters in a node and parent-pointer based traversal (for example, Hapala's method [Hap+11]) can optionally encode a parent index in a given node.

Hwang et al[Hwa+15] experimented with hybrid fixed/floating point representations in hardware suited for mobile computing. Watertight ray traversal with reduced precision[VAS16] experimented with reduced precision representations and compression for hardware acceleration structures.

Liktor et al developed a memory layout and node addressing scheme and mapped it to a custom hardware design for fixed-function ray traversal[LV16].

Benthin et al developed a Compressed-Leaf BVH[Ben+18] - a hybrid model BVH which aims to maintain almost the performance of an uncompressed BVH, but the space usage of a compressed BVH.

2.6. GPU architectures

GPU architectures impose a varied set of additional challenges on top of optimising ray traversal for MIMD or SIMD architectures - ie CPUs.

GPUs are built around an SIMT architecture. In practice this means that many threads of execution (typically 32) are bundled together into a group (called a *warp* in NVidia parlance) with a single instruction fetch and decode unit; This means that every thread must execute exactly the same instruction, and any conditional divergence results in threads being held idle whilst their siblings take alternate paths through code. Divergent code and idle processor units is said to reduce *occupancy*, and increasing occupancy is one of the primary optimising tasks.

GPUs also tend to have a more complex memory hierarchy than CPUs, and require the programmer to explicitly manage this. CPU cache architectures involve a coherent shared cache and common memory across processing resources; Whilst naïve access to memory will certainly result in poor performance, the hardware will allow memory access to any location in storage from any processing core. A GPU, on the other hand, is a non-uniform memory access (NUMA) architecture - it explicitly has different categories of memory, with varying degrees of sharing between processing resources. For example, commonly a thread has access to its local storage, accessible to it only; a shared memory, accessible to all threads in a warp; and the global memory, accessible to all processors in the GPU. As a memory becomes more shared, contention increases. Host memory and disc could also be considered to fall into this hierarchy, at much greater access cost. Optimising for host memory from a GPU is beyond the scope of this research, but certainly relevant to ray tracing research in general.

In general, GPU architectures have not had memory caches, instead using latency hiding techniques. Latency hiding is a method employed at the hardware level, which involves having many streams of work available. Whilst latency hiding is used at the CPU level to a degree (for example, Intel's *hyperthreading*), it is used extensively at the GPU level. When a memory transaction causes processing to stall, a hardware based scheduler swaps the current warp with one that has had its memory transaction completed. Thus to ensure adequate occupancy, sufficient work must be queued. The amount of registers required by a certain piece of code is significant, as it dictates how many threads can be held idle waiting for memory. Recent GPU designs are now also moving to also having caches available, thus making memory optimisation more complex.

Unlike CPU architectures, GPU architectures have a large degree in flexibility between revisions. Whilst the current range of contemporary Intel CPUs retain binary compatibility with CPUs from the 1980s, GPU makers are free to re-architect their designs at any time. Back- and forward-compatibility are maintained using software interfaces, rather than hardware standards. The software interfaces between the programmer and the hardware ensure that code is practically always JIT compiled to some degree. Due to this flexibility, it is worth carefully evaluating the tools available, and how performance varies with software revisions. It also carries the risk of tuning one's design too heavily to one exact hardware implementation, as well as imposing a smokescreen as to what is going on at the hardware level.

Ultimately optimising this work for a GPU will involve optimising memory bandwidth usage and occupancy. There are a few seminal works that provide great insight into ray tracing on GPUs [AL09] [AK10] [LKA13].

The proposal of Aila and Kerras[AK10], whilst simulating a hypothetical architecture, provided many interesting insights into GPU architectures and memory traffic of batching algorithms. The future work section of this article contains many interesting strategies - in particular adapting stackless traversal and wider trees to their BVH-treelet model.

NVidia have introduced memory compression in the Pascal architecture[NVIa]; it is claimed that memory compression improves the effective bandwidth from L2 cache to main memory by 50%. Some investigation may be warranted into the effects of this feature.

Majumdar et al[Maj+15] devised a taxonomy for different classes of GPU kernels -

- A. Compute-Bound Kernels
- B. Memory-Bound Kernels
- C. Balanced Kernels
- D. Kernels That Do Not Scale

Rather than just trying to optimise kernels based on statistics and hardware capabilities directly, they postulate that classifying them can help to guide optimisation.

Saremi devised a thorough and vendor independent analysis toolkit for GPU performance evaluation[Sar18].

3. Implementation

3.1. Overview

The research involved taking the existing *RayCrawler* algorithms of Gasparian and Bikker [Gas16], which were targeted to a single-threaded CPU environment, and adapt them to a GPU environment. The SIMT nature of GPU platforms made effective parallelisation a challenge; The existing RayCrawler was optimised to maximise cache performance in a single-core CPU execution. In particular, the challenge of avoiding divergence (and thus low occupancy) was significant.

The intent of the original RayCrawler, as well as this research, is to investigate ray batching particularly as it pertains to memory and cache performance (see section 2.2). The overarching strategy is to reduce memory traffic by *batching* rays in a collection (or *bucket*), and dynamically creating path segments by traversing large numbers of rays against subsets of the BVH; thus amortising the cost of the BVH memory transaction across many rays. As different rays take varying paths through the tree, diverging rays can be parked in a bucket, and then picked up at a later time to take their desired alternative path, with other similarly pathed rays.

The experimental codebase was built on top of an existing cpu-based BVH builder and simple GPU-based path tracer from earlier research. A working and tested CPU-based path tracer with a robust SBVH [SFD09] builder and traversal code is available as a starting point.

The core RayCrawler algorithms for top-level BVH construction (see section 3.2) and traversal (see section 3.3.1) were imported and adapted from the existing RayCrawler codebase. The GPU treelet creation and traversal mechanism was adapted from Lighthouse2[Bik21].

Only ray generation, BVH traversal, geometry intersection, and shading code were implemented for execution on the GPU. All remaining execution stayed on the CPU - in particular, the various BVH construction mechanisms.

3.2. BVH Generation

All BVH generation takes place in CPU code. After the geometry is loaded, the BVH generator produces a BVH2. This is compressed into a a BVH4 form. The BVH4 in turn is converted into subtrees in one of two formats -

- *GpuBvh4* subtrees to be used as the treelet format[YKL17], suitable for Lighthouse's traversal code.
- A *TopLevelNode* format root BVH, which maintains the original RayCrawler's format.

The algorithm for generating the top level BVH is taken directly from RayCrawler, and the code and data structures are largely unmodified.

The top-level BVH is constructed by starting at the root of the the original BVH and copying nodes recursively until the algorithm stops and creates a treelet. A treelet remains stored in a seperate data

structure, and is referenced in the top-level BVH at the point at which it is pruned. This stop occurs when one of two conditions is met:

1. The current node contains a child with a *treeletSize* smaller than a tuneable threshold, controlled by variables MIN_TREELETSIZE and TREELETSIZE. TreeletSize refers to the total number of nodes under and including the node in question.

2. The current node has a depth of 16.

The hard limit of constraint 2 is due to a 64bit integer stack being used to track traversal progress through the tree, requiring 4 bits per level. The details of this are described as part of the top level traversal algorithm (see 3.3.1). The parameters of constraint 1 are experimental variables which may be varied during during testing.

Whenever a treelet is pruned from the root BVH, a so-called *bucket* is created and associated with the root of the treelets. Each bucket contains two RayQueues, referred to as the entry and exit queues; The entry queue contains rays waiting to enter the treelet, whilst the exit queue contains rays waiting to leave the treelet and continue traversing the top-level tree. The root node of the top-level BVH also has a standard bucket associated with it; this is explained in section 3.3.1.

A separate structure *BucketState* is maintained for the purpose of bookkeeping; This tracks the total number of rays per bucket, and is designed for quick scanning.

3.2.1. Main work loop

A *streaming* model of kernel execution was implemented - that is GPU kernel invocations are long-running, and do not rely (heavily) on the CUDA implementation to schedule specific work. This is as opposed to, for example, coupling a kernel invocation to the lifetime of one ray, or even more granular models.

The streaming model was closely coupled to the ray-batching model itself. Earlier research has considered this issue in some detail[AL09]. In some ways, this work extends the wavefront[LKA13] model of path tracing, albeit using queued path segments rather than generating new ones.

All active kernel invocations are equal during path tracing - they use the same entry point, and will independently pick one of several tasks to perform based on current state.

- New Ray Generation: Not all rays are generated in one go; Instead rays are generated on demand to keep enough live rays in the system to prevent the GPU becoming starved for work (see 3.2.2).

- Top Level Traversal: Top Level traversal will progress a group of rays through the master tree (see 3.3.1).

- Treelet Traversal: Take rays from a treelet's entry queue and traverse them through the treelet, as well as possibly intersecting with geometry. When traversal through the treelet is completed, rays are placed back in treelet's exit queue for further top-level traversal. (see 3.3.3).

- Shading: Finished rays - that is rays that have had their intersection query answered - are shaded using a simple shading method and possibly terminated and reused if their path is complete (3.7).

At any given time, all active rays are either stored in buckets, or being traversed. Once a ray is removed from a bucket, and until it is placed in another bucket (or terminated), it is owned by the thread that removed it; When outside a bucket, the ray is stored (either directly by value or indirectly by reference) as a local variable in the thread's stack. This avoids any kind or concurrent access to

rays; Locking is performed at the bucket level, once a ray is owned by a thread, it can assume exclusive access to it.

3.2.2. New Ray Generation and Next Work

New ray generation is driven by the next-work generator, which tracks progress through the render and triggers new rays to be created as required. This involves acquiring an available ray structure, resetting its state and setting its initial origin and trajectory (this process is analogous to launching a primary ray in a whitted-style[Whi79] ray tracer). Finally, the ray is enqueued in the root bucket ready to begin its traversal through the BVH.

The total number of primary rays generated is defined by the number of pixels in the output image, multiplied by the number of samples per pixel(SPP); When generating primary rays, the system can either work in pixelFirst or frameFirst mode.

1. PixelFirst: Generate all primary rays for a given pixel contiguously, then iterate to the next pixel.

2. FrameFirst: Traverse all pixels, creating one ray per pixel. Repeat until SPP rays have been generated for every pixel.

Rarely will all new rays be generated in one go; Instead, blocks of rays will be generated whenever the number of active rays falls below the tuneable parameter **ACTIVE_RAY_LOWATER** and ceases when it reaches **ACTIVE_RAY_HIWATER**. The next-work generator is warp aware, and will issue blocks of work in units of warps (that is, 32 threads).

New ray generation ceases when the required number of rays (ie SPP) have been generated for each pixel; All threads will terminate when all rays have been generated, and there are no more active rays remaining in the system.

"Boiling down" the image - that is taking the average energy per sample for every pixel's total accumulated energy - to produce the final colour value is performed in a separate kernel invoked after the main path tracing process is completed. One instance of this kernel is invoked per pixel.

3.2.3. Next Bucket Selection

When choosing a next bucket, one of three algorithms will be used; This will result in a selected bucket and choice of either the entry or exit queue.

- Random: Pick any bucket at random

- Next: Choose the bucket following the last chosen bucket (on this warp).

- Mostfull: Scan all buckets and choose the bucket (and thence entry or exit queue) containing the largest number of rays.

The *Mostfull* selection is the only non-trivial implementation. The whole BucketState array must be scanned. Each active thread in the wrap will scan a subset of the array, storing a local decision. When scanning is complete, results across the wrap are merged using a butterfly reduction[Lui14] to give one final result.

Next Bucket selection is a heuristic rather than a perfect deterministic decision; Bucket sizes change frequently in parallel. Scanning of BucketState is performed without any locking. It is possible to

select a bucket based on the number of rays contained, and have this change by the time it comes to consuming the rays; Conversely, earlier entries already scanned might change before completing the scan. Additionally, multiple warps scanning BucketState at the same time will likely choose the same bucket which might lead to sub-optimal results.

The *BucketState* array is maintained using a minimal memory footprint - 3 x 32 bit integers per bucket. This structure is fixed - it is allocated along with the buckets during top-level BVH generation. The ray counts stored within are modified using atomic operations to avoid a spinlock.

Once the bucket is chosen, the larger queue (that is, entry or exit queue) will determine the actual task to be performed; If it is the bucket's exit queue, then rays contained therein will be traversed through the top-level tree. If it's the entry queue of the root bucket, they will be shaded (and possibly terminated). Finally, if it's the entry queue of any other bucket, the rays will traverse the treelet associated with the bucket.

3.3. BVH Traversal

3.3.1. Top-Level Traversal Overview

Methods that traverse rays through an acceleration structure frequently maintain traversal state independently of the ray structure itself. Through both recursive and stackless traversal methods, the current traversal state (typically the current closest intersection and remaining parts of the acceleration structure to visit) are encoded in local variables.

A ray batching system decouples a traversal through an acceleration structure from any function calls and associated local state. As a ray may be enqueued and dequeued in a bucket, it must also maintain enough state to restart the traversal through the BVH without repeating or skipping required work.

The existing top-level traversal algorithm from RayCrawler was largely maintained. When a ray is dequeued from a bucket, the bucket in question implies its current node in the top-level BVH.

Each ray maintains a stack, referred to as a *bitstack* denoting which node's children it still desires to visit in the path from the current node back to the root. As there are four children for each node, each depth level requires 4 bits, and thus a 64 bit stack allows for a top-level BVH of at most 16 depth levels. The least significant 4 bits of the stack represent the 4 children of the root node, with each following 4 bits representing each following depth level, respectively.

When visiting a new node, the ray is intersected with the children's bounding boxes. The result of these up-to 4 intersections are stored in 4 bits (referred to as the *interestMask*), and written to the current position in the stack. The ray will then descend into the child with the closest intersection, setting its bit in the stack to zero to prevent re-entering this node in the future.

When a ray completes processing a given node and needs to step up in the tree, it can quickly scan the bitstack for the next level with bits set to 1, representing unvisited nodes of interest. Each node in the top-level BVH maintains an array of (up to) 15 ancestor pointers, which are set during construction time. Thus, stepping up to the next node requiring work can be achieved in constant time, and without touching any intermediate nodes.

When encountering a treelet-root node (and its associated bucket), the ray may choose to enter it. In which case, the ray is enqueued in the bucket's entry queue, and this ray's current traversal segment is completed.

Newly created rays are placed in the root bucket's exit queue - as the exit queue of a given bucket contains rays wanting to enter the top-level BVH at the associated node. Having a standard bucket a the root node means that special cases are not required when traversing the root node, or appraising which bucket to process next.

Rays that have completely finished traversal are placed in the root bucket's entry queue, and are ultimately picked up to be shaded, and possibly terminated.

3.3.2. Top-Level Traversal Parallelism and Algorithm Variations

The original RayCrawler traversal algorithm was adapted for use on a GPU in a relatively straightforward manner. As the majority of traversal concurrency issues are handled within the RayQueue mechanism, the core traversal code remained structurally similar; Once a ray is dequeued from a bucket, it is owned by the dequeuing thread, and thus the thread has exclusive access to said ray. During traversal, access to other data structures is read-only. Thus the single threaded version can be made to work largely as-is on a gpu.

Various minor elements required reimplementation - such as replacing parts reliant upon Intel AVX instructions. The GPU code was designed to allow for compilation on a CPU (albeit in a single-threaded fashion), so some degree of functional testing comparing the old to new implementation was possible.

A top level traversal cycle begins at a given input bucket chosen by the nextWork algorithm. It will continue consuming rays from the input bucket's exit queue until it is empty. The bucket may also be refilled by other active warps depositing rays in parallel.

As memory bandwidth optimisation is central to this research, for each BVH node fetched from RAM, we want to intersect the maximum number of rays. Ideally, a full warp of threads (with one ray per thread) would intersect each BVH node. One of the core challenges with designing these algorithms is that, as rays diverge and are no longer interested in following certain paths through the BVH, this cost amortisation drops.

The straightforward adaption of the original RayCrawler algorithm was referred to as the NAIVE algorithm; It will start all threads at the same bucket (as chosen by the Next Bucket algorithm in use), each dequeuing one ray from the exit queue, and then independently traversing until a thread encounters a subtree its ray wishes to visit. The ray is then enqueued in the subtree bucket's entry queue; alternatively a ray might complete traversal, which means enqueuing in the root bucket's entry queue. This process will continue until the starting bucket's exit queue is completely emptied. Once rays are dequeued from the starting bucket, the bucket remains unlocked during traversal. Thus, other warps may enqueue rays whilst traversal is taking place. The NAIVE algorithm allows all threads in a warp to completely diverge; This keeps the threads far busier at the cost of more incoherent memory access.

The NAIVE_INTERLOCKED algorithm is largely identical to NAIVE, however all threads will synchronise after their ray has been parked in a queue, before restarting the process by dequeuing a new ray from the starting bucket. In other words, they will resynchronise before starting a new ray's traversal, and reduce accesses to the starting RayQueue as many rays can be bulk-dequeued in one transaction, at the cost of threads sitting idle waiting for the longest running thread in a warp. Once traversal has begun, the threads in the warp operate independently allowing each ray to take its own path through the top-level BVH.

Improved algorithms for traversal were designed, but never fully implemented. These are outlined in section 6.1.

3.3.3. Treelet Traversal

The treelet traversal mechanism was taken from from Lighthouse2[Bik21]; This is a highly optimised traversal implementation for GPUs. Each thread will dequeue rays from the treelet's bucket's entry queue, traverse them, and enqueue them in the exit queue. No batching is implemented within treelet traversal, so traversal state can be held completely within local variables; The ray's top-level traversal stack remains unmodified by the treelet traversal code.

The given warp Will continue consuming rays from the bucket's entry queue until the entry queue is empty. Other warps may deposit inbound rays whist the given warp is traversing against the treelet.

3.4. Rays

3.4.1. Ray Storage

The ray structure was used to store ray traversal and energy state. The ray was divided in half to store state needed for traversal (*Ray*, see listing A.1) and state only needed when shading (*RayExtra*, see listing A.2). The intent was reduced memory traffic required during the traversal phase.

When in INDIRECT mode (see section 3.4.2), the indices of the master array of Ray and RayExtra structures are maintained in parallel, so the same index can be used to access the corresponding structures. When in DIRECT mode, the rays are copied, so there is no index to match. In this case, the index to the RayExtra array is stored as a member of Ray.

There are a fixed number of Rays and RayQueue objects in the system, allocated before path tracing commences. As there is no "malloc" available in the environment, a mechanism is required to track idle rays and RayQueues.

The *SpareQueue* holds unused RayQueue objects. The RayQueues themselves remain empty whilst held in the SpareQueue, and instead are linked together using the single-linked list mechanism of the RayQueue. Access to the SpareQueue is protected by its own spinlock.

The *IdleQueue* is used to store any idle rays - which includes all rays when the system is first started. When in DIRECT mode, rays are stored by value and copied; so the effective storage for all rays is in RayQueue objects, or local variables whilst traversing. When in INDIRECT mode, all rays are stored only in a large buffer, and referred to by index. In this case, the IdleQueue is used to store indices of arrays currently not traversing, for quick access when starting new rays. As a byproduct of using the RayQueue mechanism, Rays are stored in the IdleQueue in approximate LIFO order.

3.4.2. RayQueues Overview

The implementation uses RayQueues as both an allocator and storage mechanism for rays. Two models of ray storage were implemented - DIRECT and INDIRECT.

DIRECT storage holds the ray objects by value in the RayQueue. When an enqueue or dequeue is required, the ray struct is copied by value. When in INDIRECT mode, the ray objects are allocated in one large block and referred to by array index when stored in a queue (thus never copying them). Terminated ray objects are reused in place.

RayQueues are implemented as a singly linked-list of arrays of rays. That is, each RayQueue could hold a fixed number of rays, and if additional space is required, an additional queue object can be

attached to the end of the existing queue.

The original intent was to use only a ring buffer, implemented as an array; This would allow FIFO queuing. This quickly became impractical however, as it quickly became clear that a fixed size array would fill quickly or have to be so large that most of the system memory quickly became exhausted - especially when multiplied by the number of RayQueues in the system.

Thus, a hybrid solution was taken modelled on the original cpu-based system. Individual ring buffers remain as the base unit of storage, but a can be chained (and un-chained) in a singly linked list to allow for growth or contraction as required. As only a singly-linked list is used, pure FIFO queuing is no longer provided.

3.4.3. Linking and Delinking RayQueues

Rays are both enqueue to and dequeued from the front RayQueue object. Only this object is permitted to be non-full; all queue objects further back in the list must be full.

Frequently, the number of rays to be enqueued to a given RayQueue will exceed the available capacity, thus an additional RayQueue object will be added to the front and the previous front will be pushed back. This transaction may require a split insert - inserting enough rays in the previous front RayQueue, and the remainder in the new front RayQueue.

3.4.4. Enqueuing and Dequeuing

When accessing a RayQueue, any number of interlocked threads in a warp (that is, up to 32) may wish to complete a transaction. Initially, the number of active threads wanting to access the queue is tallied, and thus the number of rays in the transaction is known. Enqueue transactions against a RayQueue thence involve a two step process, based around the concept of warp-aggregated atomics [Adi14]:

- Choosing a leader thread. This thread handles locking / unlocking the spinlock protecting queue (see 3.5), as well as updating the indices of the RayQueue to allocate space, and as determining the base index at which to start the insertion.
- Each thread then calculates its own offset from the base index in the array, based on its thread id's position in the set of active threads. Each thread then performs its own write into the queue, using this index. The write is either INDIRECT (using an index to the master array of rays) or DIRECT, copying the ray into the RayQueue.

As storage within the queues is guaranteed to be contiguous, it is a matter of each thread determining its offset point from the start of the queue; It need not worry about encountering empty slots and having to search for a ray.

When more rays are to be enqueued than will fit in the RayQueue object, a new RayQueue is obtained from the spare list, and singly-linked to the existing RayQueue. This new RayQueue becomes the front queue object. If no spare queues are available, the system will fail, as this is beyond the capabilities of the current implementation. Now, sufficient space is available a split enqueue is required.

When a split enqueue is required the algorithm is complicated slightly. The rays (and thus threads) are split into two groups; Enough rays to fill the space in the old front RayQueue, and the remainder which go into the new front RayQueue. The transaction is executed twice in parallel, with a leader elected for each RayQueue.

Dequeuing is the same process in reverse, possibly involving freeing up a RayQueue object and returning it to the spare queue.

The current setup does not allow a single enqueue or dequeue transaction across more than two RayQueue objects, as each thread will only enqueue or dequeue a single ray, and the size of the queues is always at least 32 (that is, the number of threads in a warp).

3.5. Locking and Parallelism

Mutual exclusion was required to protect access to shared data structures - predominantly RayQueues - and this required solving two kinds of locking; inter-warp and intra-warp locking.

Historically, GPU architectures have one instruction pipeline and program-counter per Warp. Recent CUDA platforms have changed the program-counter to be per-thread[nvi17], thus allowing complete divergence of threads' program location.

Inter-warp locking required a mutual exclusion mechanism; This was implemented using CUDA's atomicCAS() to implement a simple spinlock. For simplicity, a 3 spinlocks were used to protect all shared storage data (as opposed to say - a spinlock per bucket). These locks protected all the active buckets, the idle queue and the spare queue. Deadlocks were challenging to avoid with a more granular locking structure, as access to buckets is effectively ordered arbitrarily.

As much as possible, threads within a warp were kept from diverging. When accessing a shared resource the intent was for as many threads as possible to access it together, as a warp-aggregated atomic transaction[Adi14]. The pattern adopted was to divide the active threads into as many groups as the number of distinct resources required (sometimes only 1 group was required) and for each group to elect a leader. In the implementation, this could split into as many as 4 groups - where rays were being enqueued to two different queues, and each queue required linking a new RayQueue to expand its capacity.

When multiple thread groups are required the *this* pointer of the objects to be accessed was used as a key to group the active threads together and elect a leader within each group via CUDA's __match_any_sync() mechanism.

Where possible, synchronisation between threads within a warp was performed using the CUDA mechanisms such as __match_any_sync() and __shfl_xor_sync() in preference to using shared data structures with locks. This avoids regular and atomic traffic to shared memory and requires fewer processor instructions[Har14].

When a ray's path is completed, its accumulated energy (ray.E) is atomically added to the shared accumulator.

3.6. Occlusion v. Closest hit

Next Event Estimation (NEE) was to allow for two kinds of ray queries against the BVH. As NEE queries require only an occlusion query, these rays could be terminated early upon an intersection with the geometry, increasing divergence. NEE can be configured on, off, or in half-half mode; The latter means one half of the frame is rendered with NEE, the other without.

3.7. Shading

A simple shading routine was reused from earlier research. It implements diffuse and specular reflection, and transmission. Additionally, it implements Next Event Estimation as described in section 3.6.

Laine et al[LKA13] make reference to the cost of shading code not being significant enough to warrant breaking into streams of work. As such, very little focus was placed on shading; only enough to provide the functionality required for the rest of the system.

3.8. Random Number Generation and Seeding

Random numbers are seeded at start of crawl per thread, using the thread id and the current pixel id and sample number. From that point the RNG's state is local to every thread. CUDA's *clock()* function was considered as a random number seed rather than tracking state through the code. However this would remove the option of repeatability and determinism of explicitly seeding the RNG so was discounted.

The RNG used is *Xorshift RNGs for small systems*[Bow14] based on XORShift[Mar+03].

3.9. Debugging Tools

To help catch bugs, various self debugging mechanisms were devised. In particular, the parallelism model dictated that as one leader thread would modify a shared structure on behalf of many threads in a warp it was imperative that they were in fact intending to access the same structure. For example, when a set of n threads would access the same RayQueue, all threads would use the *_match_any_sync()* function to compare their *this* pointers (that is, the RayQueue in question). If the number of active threads does not match the number matched by *_match_any_sync()*, an error has occurred and an assert is fired.

4. Experiments and Results

4.1. Overview

Experimentation was focussed on testing three areas of the system - The size of the top-level BVH (relative to the size of treelets), the number of active rays in the system and the size of each RayQueue object's contiguous storage array. DIRECT v. INDIRECT mode (see 3.4.2) was also tested at each step as it changes the memory access pattern significantly.

Due to limitations in the implementation, it was ultimately decided to run the system only with a single warp. Unfortunately, this limited the ability to stress the system.

Two scenes were used - *Crytek Sponza* with 262268 triangles, and *Dragon* with 871198 triangles. Both scenes had a simple emissive plane added (made of 2 triangles) to act as an area light. All images were rendered at a resolution of 1280 x 1280, with an SPP value of 2. Dragon is an open scene - with a non-trivial possibility that initially launched rays might miss the object all together. Whilst Crytek Sponza was not completely closed, the placement of the camera means that all rays launched from the camera should intersect with geometry at least once.

Experimental parameters are set at compile time via the file *tuning.h* or via the compiler command line. This allowed the script *bulk.py* to generate a mass number of runtime instances and run the system against a large number of parameter sets, as well as automatically collecting the results.

The test GPU hardware was a NVidia GeForce RTX 2080 - with 8gb of ram. During experimentation, care was taken to ensure that minimal external factors could add variance to the results. The system was tested on a machine running a standard desktop operating system (*Ubuntu Linux 20.04.2 LTS*) and as such parts of the system running on the CPU were in contention for resources with other processes. However, testing was performed with the system's GUI shut down - in the so-called text mode or in *runlevel 3* in Unix parlance. The result being that the GPU was effectively dedicated to CUDA clients rather than shared with GUI rendering tasks; the system under test was the only CUDA client in use at the time. This was confirmed with the *nvidia-smi* utility before commencing a testing run to ensure no other processes had open CUDA resources. Only code running on the GPU was measured in the following tests.

4.2. CUDA Profiling

For every parameter set, the system was (optionally) run twice. Once directly, where the run-time is measured, and if applicable under the CUDA profiler, where selected hardware statistics are recorded. The applicable CUDA profiler *nv-nsight-cu-cli* was used to capture the memory access hardware statistics group listed in table 4.1. Hardware profiling was used sparingly as it could easily take over 45 minutes to run one instance of a parameter set.

Parameter Name	Unit
Memory Throughput	Gbyte/second
Mem Busy	%
Max Bandwidth	%
L1 Hit Rate	%
L2 Hit Rate	%
Mem Pipes Busy	%

Table 4.1.: Parameters captured from the CUDA profiler

4.3. Initial Run and Tree-Size Testing

For the initial run, a range of parameters was selected, which are listed in table 4.2. Whilst a vastly larger combination of parameters was possible, the restricted set was chosen to make testing practical. The effect of various parameter changes were tested early to eliminate variables where possible - in particular, no measurable difference between the NAIVE and NAIVE_INTERLOCKED modes was observed.

Parameter Name	Values
SCENE	Dragon, Sponza
TOPLEVEL_MODES	NAIVE
NEXTWORK_ALGO	PIXELFIRST, FRAMEFIRST
BUCKETSELECT_ALGO	MOSTFULL
NEEMODE	ON
REGCOUNT	64
DIRECT	INDIRECT, DIRECT
NUM_WARPS	1
SPP	2
TREELETSIZE	200, 800, 1600, 2000
RAYCOUNT	1024, 10240, 1048576
MAXDEPTH	7
BVH_MAXDEPTH	15
SCRWIDTH	1280
SCRHEIGHT	1280

Table 4.2.: Parameters used during initial testing run

This parameter set resulted in 96 unique combinations to be run.

Four best results were chosen from the initial run, that is the best time for INDIRECT and DIRECT mode for each scene. These are summarised in table 4.3. The complete results of the initial run are listed in appendix C.

Scene	DIRECT?	NextWork	treelet	rayCount	Render Time (sec)
Dragon	DIRECT	FRAME	800	10240	31.3964
Dragon	INDIRECT	PIXEL	2000	1024	59.9774
Sponza	DIRECT	PIXEL	200	1024	151.381
Sponza	INDIRECT	PIXEL	800	1024	164.328

Table 4.3.: Best results from initial run

4.4. Ray Count Experiments

Given the best performing parameter set from the initial run, experiments were performed varying the number of concurrently active rays. The 14 values of 2^8 through to 2^{21} were used for testing; The upper bound found as performance stabilised at this point. The complete results of the RayCount testing run are listed in appendix D.

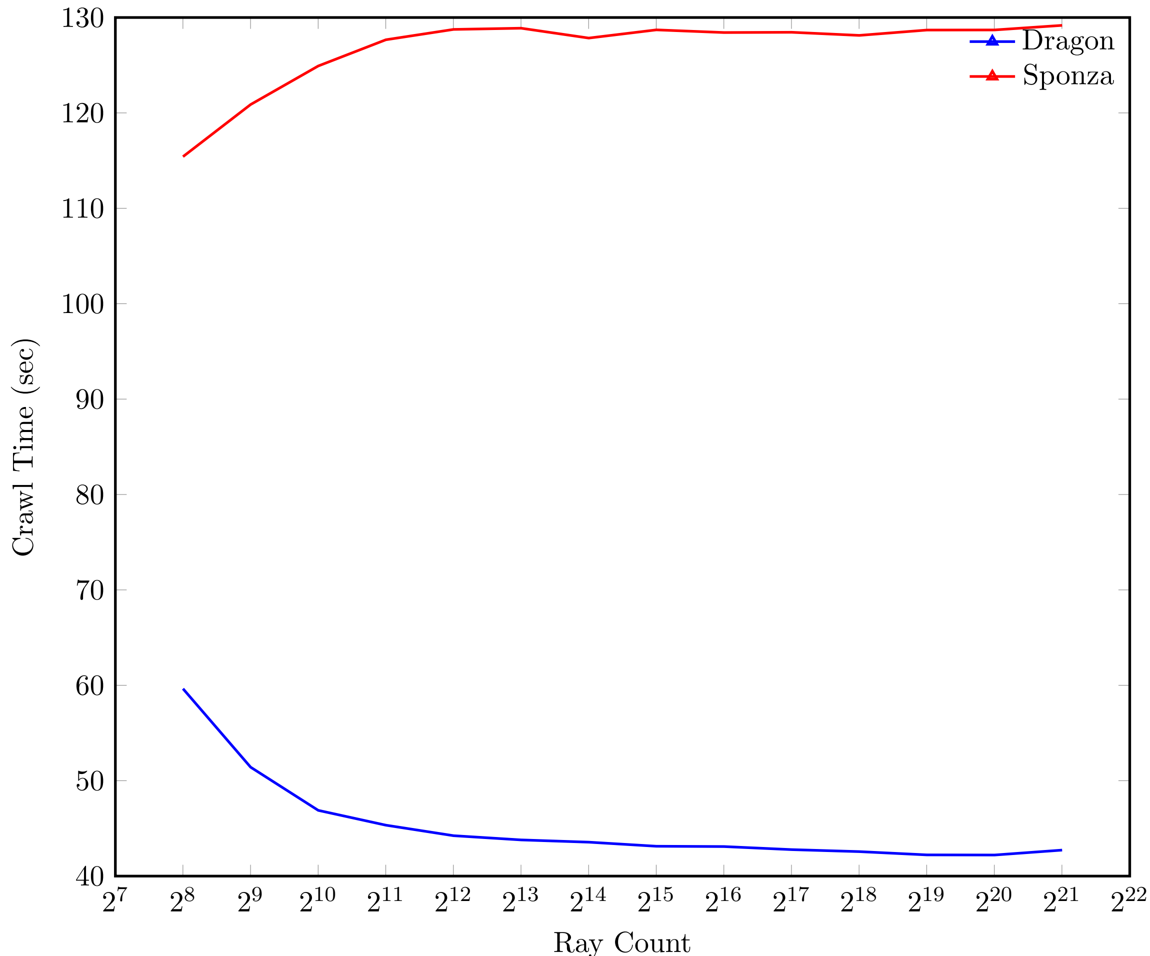

Figure 4.1.: Traversal times across both scenes with varying numbers of active rays

A TREELETSIZE parameter of 800 was used for both scenes, and in effect, the same BVH was used for all runs through a given scene. The details are listed in table 4.4. Importantly, despite having the same construction parameters, the resultant characteristics of the top-level BVHes varied greatly.

Scene	Top Level Size (nodes)	Treelet Count	Average Treelet Size (nodes)
Dragon	95	71	2178
Sponza	11	8	5927

Table 4.4.: Nature of BVH size distribution between scenes

4.5. Queue Size Experiments

Using a ray count of 1048576, and again a TREELETSIZE of 800, various RayQueue sizes were investigated. The results are summarised in figures 4.2 and 4.3. These tests were performed for both DIRECT and INDIRECT mode, as this causes a notably different memory access pattern. The

complete results of the QueueSize testing run are listed in appendix E.

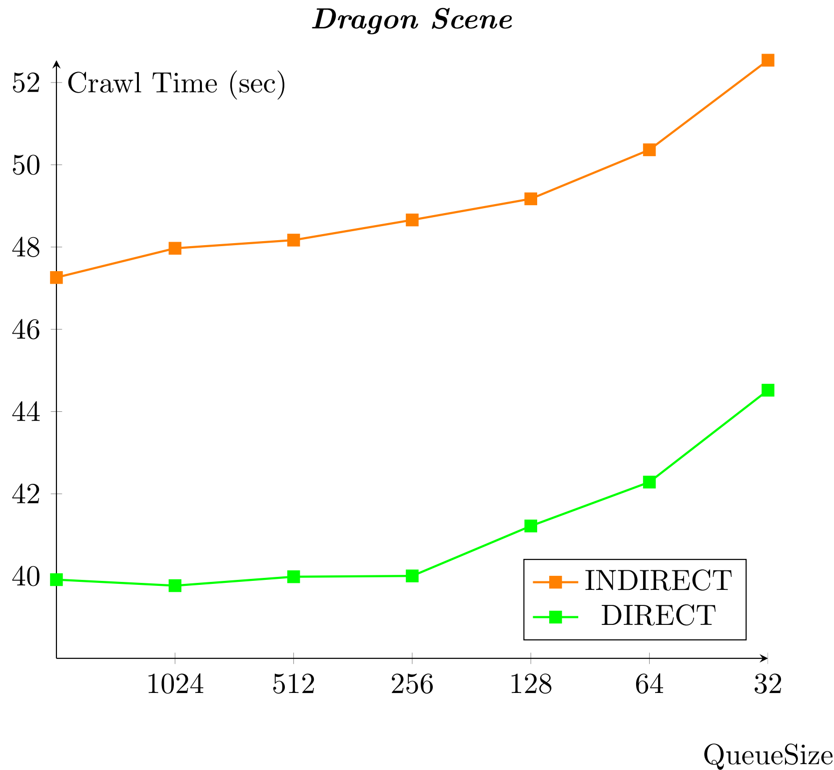

Figure 4.2.: Comparison of performance of varying queue sizes for scene Sponza

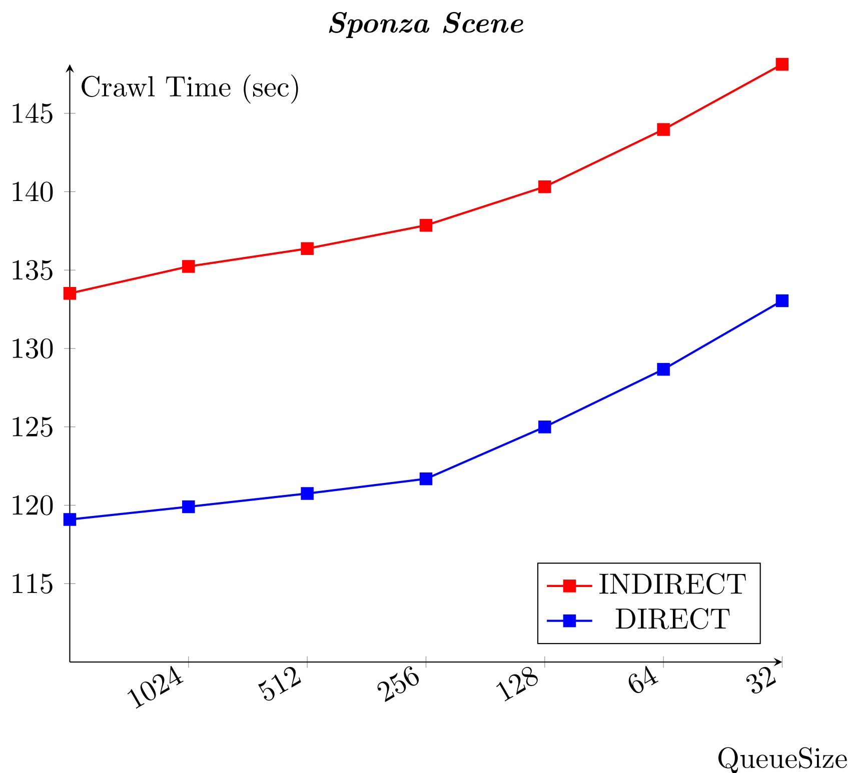

Figure 4.3.: Comparison of performance of varying queue sizes for scene Sponza

The range of queue sizes was determined by a hard minimal limit - the system can transact with at most 2 RayQueues in a single transaction, so 32 (being the max number of threads in a RayQueue transaction) was used as a minimum to guarantee that a transaction can never cross more than two queues. The Upper bound was determined by memory limitations on the hardware in question.

Every test run was also profiled. The QueueSize test showed average hit-rates on the L1 cache of 61.7325% and L2 cache of 74.6225%

5. Discussion

Examining the memory usage statistics in table E, it seems that the lack of extensive parallelism (due to only using a single warp) is failing to exercise the hardware in a meaningful way; The largest contributor to improvements in cache performance is the selection of scene.

Whilst the top-level builder algorithm has various parameters to control the thresholds for creating and sizing treelets, even with identical parameters the distribution of geometry within a scene causes a marked difference in the final top-level/treelet BVH layout. For example, with a TREELETSIZE parameter of 800, the algorithm generated 71 treelets for the *Dragon* scene and 8 for the *Sponza* scene. These two scenes have triangle counts within the same order of magnitude.

The resultant layout of the overall BVH has a marked effect on the rendering performance. This includes factors such as time spent in each portion of code and the number of potential batching points (i.e. buckets) as well as the potential number of simultaneously live rays. This in turn means that the division of time varies between treelet traversal using an existing traversal mechanism already highly optimised for GPUs (Lighthouse2's traversal code) and the top-level traversal implementation from the existing system designed and tuned for CPUs.

The most striking performance difference came from DIRECT vs. INDIRECT mode. This mode change causes a number of important changes to the code.

Ray enqueue/dequeue transactions will use only an index in INDIRECT mode, whilst the whole ray is copied in DIRECT mode. This leads to larger memory transactions in DIRECT mode, however it effectively compacts all the rays in use into one (or two) coherent block(s) of memory within the RayQueues.

When traversing, the ray is copied to a thread's local storage in DIRECT mode, whilst in INDIRECT mode the ray stays in shared memory (only the ray's index is copied locally). This leads DIRECT to having reduced shared memory traffic, as well as making INDIRECT access incoherent - There is no reason to believe that any two rays queued in a RayQueue consecutively will be allocated coherently in the master ray array. The ray structure is accessed frequently during traversal, causing these differences to be significant.

6. Conclusions And Future Work

This research has demonstrated that it is possible to implement the RayCrawler model in a GPU environment. However, the current implementation does not demonstrate a practical system as it exploits only a fraction of the resources available in a GPU. The limited parallelism in the current implementation appears to have failed to exercise the hardware in any meaningful way; during the QueueSize tests, whilst cache utilisation gave promising numbers (60%-70%), the overall memory traffic was low; Despite this, the performance was vastly inferior to many contemporary ray tracing systems.

Various useful results were observed, in particular showing improved performance with more localised and coherent memory access; Reliable correlations in performance characteristics between DIRECT and INDIRECT modes in particular showed that more localised coherent memory access yielded better performance, even at the cost of larger memory transactions (that is, constantly reading and writing the entire ray structure). Further increasing the coherence of memory access - by enlarging the RayQueue's QueueSize giving a larger block of memory to work in before needing to link-in a new RayQueue - also object increased performance. In some cases increasing the number of active rays - allowing more work to be performed on the same bucket before moving on - further increased performance.

It is also noteworthy that the performance trend in the RayCount test went in differing directions as the number of rays in the system increases. Despite having the same BVH construction parameters, and a number of geometric primitives of the same order of magnitude, the resultant BVHes for both scenes have dramatically different characteristics, as shown in table 4.4. It is likely that in the Sponza scene huge numbers of rays are generated, and have to be enqueued and dequeued through a relatively small number of RayQueues (in this case, having a size of 64 rays) - effectively serialising processing large chunks of rays through the system and making memory access incoherent. The Dragon scene, on the other hand, with its large number of treelets (and thus RayQueues) allowed the rays to diverge more effectively and skip larger chunks of the tree without needing to be queued.

6.1. Parallelism and Improved Traversal Strategies

Future research may expand the system to truly engage a GPU's many parallel resources. Most importantly a full parallel version, allowing an arbitrary number of warps to be launched concurrently should be able to move closer to fully exploiting the hardware.

With a full parallel implementation, further study into convergence and divergence of threads would be possible beyond the existing NAIVE and NAIVE_INTERLOCKED models. Two other algorithms to improve upon the NAIVE versions were designed, however due to time pressure they were not implemented completely.

SHORT_SEGMENT: Traverse in a similar fashion to NAIVE, but whenever encountering a bucket, enqueue all rays in this bucket. Those that wish to enter the treelet in question would enqueue in the entry queue, those that didn't would be enqueued directly in the exit queue, allowing them to skip the treelet. Then return to the starting bucket. Rays that did not want to descend into a branch of the top level BVH (referred to as 'sulking' rays) would be forced to do so in order to keep the warp

non-divergent. This as an alternative to simply terminating the whole warp when >50% of rays are dead, and re-queuing rays at the top of the tree as has been done previously[AK10].

LONG_HAUL: Continue traversing at all costs. When encountering a bucket, if a given ray wants to enter that bucket, add it to the entry queue, and replace it with one from the exit queue. Descend to a branch of the tree if and only if at least one ray wants to go that way.

Various unresolved complications exist with both of these approaches - in particular what to do when divergence occurs at an interior node - if some rays wish to descend and some do not, what action should be taken?

6.2. More Granular Measurements

Presently, only the total time taken for the system to render an image, as well as the memory access statistics from the CUDA profiler are available. Recording statistics such as the number of times rays are queued, and the number of rays entering a treelet would be very useful for further analysis. Additional, having some idea as to the amount of processing time spent in the various parts of code would be useful - that is to say comparing time traversing treelets vs top-level, ray generation and so forth.

6.3. Multi-tier Trees

Explore a n-tier hierarchy model rather than just top-level and treelets. As suggested by Gasparian and Bikker[GB17], *"Given the high performance gains for complex scenes, it may make sense to add an additional level in the BVH hierarchy for extremely large scenes where the top-BVH itself no longer fits in the L2 cache"*. This latter option may also allow for using the same traversal algorithm across all levels of the tree;

There is no implicit reason that different subsets of the acceleration structure require different traversal implementations, so long as an implementation provides the facility to pause and restart traversal - effectively freezing a ray's state. Further more, due to the extensive parallel processing capability of GPUs, further subdivision of the BVH may lend itself to improved performance with smaller subsets of the acceleration structure more localised to separate processing resources.

Finally, it would be useful to enable a mode using a single treelet only - no top-level BVH at all. This would effectively disable batching all together, and allow for a more 'straight-through' traversal method. This would provide a useful baseline case to compare performance when batching.

6.4. Multi-queuing Rays and Fairer Queuing

Consider a model allowing a given ray to exist in multiple buckets simultaneously; If the initial pass through the top-level BVH finds multiple leaves or treelets that are to be traversed by a given ray, it could be queued in multiple buckets. This will most likely involve more memory traffic and at least one extra level of indirection, as well as an atomic reference counting and maintenance of the current best intersection. It does lead to the possibility of intersecting structures when a closer intersection is already known, so methods of handling this would be required.

There is no guarantee of fairness, which leads to the possibility of rays sitting in buckets for extended periods. The system will ultimately complete them all, but this will lead to a kind of 'mopping up'

at the end of rendering a frame. Additionally the current nextBucket algorithm does nothing to stop multiple warps processing the same bucket. With the possibility of a multi-tier BVH, explore more rigidly assigning processing resources to buckets to keep the mechanism more streaming-oriented.

6.5. Dynamic Allocation

The system uses static allocation for all data structures; There is no ”malloc” in a CUDA environment. This leads to guesswork when tuning the system and potential for starvation. For example, with a very large QUEUESIZE, the QUEUECOUNT must be reduced to prevent exceeding available ram. In turn can lead to a scenario where the system fails as no RayQueue objects are available when required.

Similarly, the levels for generating new rays are fixed and created empirically. Further research could lead to smarter methods for managing these issues.

6.6. Memory Access

As memory performance is so central to the optimisation, study of the caching mechanisms will be important; Explicit pre-fetching of data into cache can improve performance. NVidia architectures do expose prefetching instructions at the PTX level[NVIb]. Some research into explicit prefetching has been performed by Lee et al[Lee+10], and they determined that explicit prefetching on a GPU can be both helpful or harmful. Revisiting this research in relation to this specific problem and on more recent architectures may be worthwhile.

7. Acknowledgements

I would like to thank my primary supervisor Dr. Bikker for his endless patience and sage advice, as well as Victor Veldstra for great ideas and feedback during brainstorm sessions. Finally, my friends and family for their ongoing encouragement and support.

8. References

[Adi14] Andy Adinets. *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics.* NVIDIA Developer Blog. Oct. 2, 2014. URL: https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/ (visited on Apr. 22, 2019).

[AK10] Timo Aila and Tero Karras. "Architecture Considerations for Tracing Incoherent Rays". In: High Performance Graphics. Saarbrüken, Germany, 2010, pp. 113–122.

[AL09] Timo Aila and Samuli Laine. "Understanding the Efficiency of Ray Traversal on GPUs". In: *Proceedings of the 1st ACM Conference on High Performance Graphics - HPG '09.* 2009, pp. 145–145. DOI: 10.1145/1572769.1572792. URL: http://portal.acm.org/citation.cfm?doid=1572769.1572792.

[AW87] John Amanatides and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: (1987), p. 6.

[BA14] Rasmus Barringer and Tomas Akenine-Möller. "Dynamic Ray Stream Traversal". In: *ACM Transactions on Graphics* 33.4 (July 27, 2014), pp. 1–9. ISSN: 07300301. DOI: 10.1145/2601097.2601222. URL: http://dl.acm.org/citation.cfm?doid=2601097.2601222 (visited on Nov. 4, 2018).

[Ben+12] Carsten Benthin et al. "Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel R MIC Architecture". In: *IEEE Transactions on Visualization and Computer Graphics* (2012), p. 10.

[Ben+18] Carsten Benthin et al. "Compressed-Leaf Bounding Volume Hierarchies". In: ACM Press, 2018, pp. 1–4. ISBN: 978-1-4503-5896-5. DOI: 10.1145/3231578.3231581. URL: http://dl.acm.org/citation.cfm?doid=3231578.3231581 (visited on Aug. 11, 2018).

[Bik12] Jacco Bikker. "Improving Data Locality for Efficient In-Core Path Tracing". In: *Computer Graphics Forum* 31.6 (Sept. 2012), pp. 1936–1947. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2012.03073.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2012.03073.x (visited on May 7, 2018).

[Bik21] Jacco Bikker. *Lighthouse 2 Framework for Real-Time Ray Tracing.* June 29, 2021. URL: https://github.com/jbikker/lighthouse2 (visited on July 8, 2021).

[BK16] Nikolaus Binder and Alexander Keller. "Efficient Stackless Hierarchy Traversal with Backtracking in Constant Time". In: High Performance Graphics. 2016, p. 50.

[Bou+07] Solomon Boulos et al. "Packet-Based Whitted and Distribution Ray Tracing". In: *Proceedings of Graphics Interface 2007 on - GI '07.* Graphics Interface 2007. Montreal, Canada: ACM Press, 2007, p. 177. ISBN: 978-1-56881-337-0. DOI: 10.1145/1268517.1268547. URL: http://portal.acm.org/citation.cfm?doid=1268517.1268547 (visited on Nov. 4, 2018).

[Bow14] James Bowman. *Xorshift RNGs for Small Systems.* Aug. 2014. URL: https://excamera.com/sphinx/article-xorshift.html (visited on July 8, 2021).

[DHK08] H. Dammertz, J. Hanika, and A. Keller. "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays". In: *Computer Graphics Forum* 27.4 (2008), pp. 1225–1233. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2008.01261.x.

[EG08] Manfred Ernst and Gunther Greiner. "Multi Bounding Volume Hierarchies". In: IEEE, Aug. 2008, pp. 35–40. ISBN: 978-1-4244-2741-3. DOI: 10.1109/RT.2008.4634618. URL: http://ieeexplore.ieee.org/document/4634618/ (visited on Apr. 24, 2018).

[Eis+13] Christian Eisenacher et al. "Sorted Deferred Shading for Production Path Tracing". In: *Computer Graphics Forum* 32.4 (July 2013), pp. 125–132. ISSN: 01677055. DOI: 10.1111/cgf.12158. URL: http://doi.wiley.com/10.1111/cgf.12158 (visited on June 15, 2018).

[FJ94] Keith I Farkas and Norman P Jouppi. "Complexity/Performance Tradeoffs with Non-Blocking Loads". In: *Proceedings of the 21st annual international symposium on Computer architecture* 22 (Apr. 1994), pp. 211–222. DOI: 10.1145/192007.192029. URL: http://doi.acm.org/10.1145/192007.192029.

[Fue+15] Valentin Fuetterling et al. "Efficient Ray Tracing Kernels for Modern CPU Architectures". In: 4.4 (2015), p. 21.

[Gas16] Tigran Gasparian. "Fast Divergent Ray Traversal by Batching Rays in a BVH". University of Utrecht, Dec. 5, 2016. 17 pp.

[GB17] Tigran Gasparian and Jacco Bikker. "Ray Batching for Faster BVH Traversal of Incoherent Rays". In: (2017), p. 10.

[GL10] Kirill Garanzha and Charles Loop. "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing". In: *Computer Graphics Forum* 29.2 (May 2010), pp. 289–298. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2009.01598.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2009.01598.x (visited on Nov. 4, 2018).

[Gla84] Andrew S Glassner. "Space Subdivision for Fast Ray Tracing". In: *IEEE Computer Graphics and applications* 4.10 (1984), pp. 15–24.

[GR08] Christiaan P. Gribble and Karthik Ramani. "Coherent Ray Tracing via Stream Filtering". In: IEEE, Aug. 2008, pp. 59–66. ISBN: 978-1-4244-2741-3. DOI: 10.1109/RT.2008.4634622. URL: http://ieeexplore.ieee.org/document/4634622/ (visited on Apr. 24, 2018).

[Hap+11] Michal Hapala et al. "Efficient Stack-Less Bvh Traversal for Ray Tracing". In: *Proceedings of the 27th ...* 1 (2011), pp. 7–12. ISSN: 9781450319782. DOI: 10.1145/2461217.2461219. URL: http://dl.acm.org/citation.cfm?id=2461219.

[Har14] Mark Harris. *CUDA Pro Tip: Do The Kepler Shuffle*. NVIDIA Developer Blog. Feb. 3, 2014. URL: https://devblogs.nvidia.com/cuda-pro-tip-kepler-shuffle/ (visited on Jan. 24, 2019).

[Has+18] Hasan Hassan et al. *Exploiting Row-Level Temporal Locality in DRAM to Reduce the Memory Access Latency*. May 8, 2018. arXiv: 1805.03969 [cs]. URL: http://arxiv.org/abs/1805.03969 (visited on Jan. 25, 2019).

[Hav00] Vlastimil Havran. "Heuristic Ray Shooting Algorithms". Ph. d. thesis, Department of Computer Science and Engineering, Faculty of ..., 2000.

[HI09] D.M. Hughes and Ik Soo Lim. "Kd-Jump: A Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs". In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1555–1562. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.161. URL: http://ieeexplore.ieee.org/document/5290773/ (visited on Nov. 4, 2018).

[Hwa+15] Seok Joong Hwang et al. "A Mobile Ray Tracing Engine with Hybrid Number Representations". In: ACM Press, 2015, pp. 1–4. ISBN: 978-1-4503-3928-5. DOI: 10.1145/2818427.2818446. URL: http://dl.acm.org/citation.cfm?doid=2818427.2818446 (visited on May 1, 2018).

[Kaj86] James T Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* 20.4 (1986), pp. 143–150. ISSN: 0897911962. DOI: 10.1145/15886.15902.

[Kee14] Sean Keely. *Reduced Precision for Hardware Ray Tracing in GPUs*. 2014. DOI: 10.2312/hpg.20141091.

[Kop+10] D. Kopta et al. “Efficient MIMD Architectures for High-Performance Ray Tracing”. In: IEEE, Oct. 2010, pp. 9–16. ISBN: 978-1-4244-8936-7. DOI: 10.1109/ICCD.2010.5647555. URL: http://ieeexplore.ieee.org/document/5647555/ (visited on Apr. 24, 2018).

[Kop+13] Daniel Kopta et al. “An Energy and Bandwidth Efficient Ray Tracing Architecture”. In: ACM Press, 2013, p. 121. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492058. URL: http://dl.acm.org/citation.cfm?doid=2492045.2492058 (visited on May 1, 2018).

[LA02] Chris Lattner and Vikram Adve. *The LLVM Instruction Set and Compilation Strategy.* 2002.

[Lai10] Samuli Laine. “Restart Trail for Stackless BVH Traversal”. In: *High Performance Graphics* (2010), p. 5.

[Lee+10] Jaekyu Lee et al. “Many-Thread Aware Prefetching Mechanisms for GPGPU Applications”. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Atlanta, GA, USA: IEEE, Dec. 2010, pp. 213–224. ISBN: 978-1-4244-9071-4. DOI: 10.1109/MICRO.2010.44. URL: http://ieeexplore.ieee.org/document/5695538/ (visited on Jan. 24, 2019).

[Lee+12] Won-Jong Lee et al. “SGRT: A Scalable Mobile GPU Architecture Based on Ray Tracing”. 2012.

[Lee+14] Jaedon Lee et al. “Two-AABB Traversal for Mobile Real-Time Ray Tracing”. In: ACM Press, 2014, pp. 1–5. ISBN: 978-1-4503-1891-4. DOI: 10.1145/2669062.2669088. URL: http://dl.acm.org/citation.cfm?doid=2669062.2669088 (visited on Apr. 17, 2018).

[Lee+15] Won-Jong Lee et al. “Reorder Buffer: An Energy-Efficient Multithreading Architecture for Hardware MIMD Ray Traversal”. In: ACM Press, 2015, pp. 21–32. ISBN: 978-1-4503-3707-6. DOI: 10.1145/2790060.2790064. URL: http://dl.acm.org/citation.cfm?doid=2790060.2790064 (visited on May 3, 2018).

[LKA13] Samuli Laine, Tero Karras, and Timo Aila. “Megakernels Considered Harmful: Wavefront Path Tracing on GPUs”. In: ACM Press, 2013, p. 137. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492060. URL: http://dl.acm.org/citation.cfm?doid=2492045.2492060 (visited on May 1, 2018).

[Lui14] Justin Luitjens. *Faster Parallel Reductions on Kepler.* NVIDIA Developer Blog. Feb. 14, 2014. URL: https://devblogs.nvidia.com/faster-parallel-reductions-kepler/ (visited on Aug. 25, 2019).

[LV16] Gábor Liktor and Karthikeyan Vaidyanathan. “Bandwidth-Efficient BVH Layout for Incremental Hardware Traversal”. In: *Proceedings of High Performance Graphics.* Eurographics Association. 2016, pp. 51–61.

[Maj+15] Abhinandan Majumdar et al. “A Taxonomy of GPGPU Performance Scaling”. In: *2015 IEEE International Symposium on Workload Characterization.* 2015 IEEE International Symposium on Workload Characterization (IISWC). Atlanta, GA, USA: IEEE, Oct. 2015, pp. 118–119. ISBN: 978-1-5090-0088-3. DOI: 10.1109/IISWC.2015.22. URL: http://ieeexplore.ieee.org/document/7314157/ (visited on Jan. 28, 2019).

[Mar+03] George Marsaglia et al. “Xorshift Rngs”. In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6.

[MB90] J. David MacDonald and Kellogg S. Booth. “Heuristics for Ray Tracing Using Space Subdivision”. In: *The Visual Computer* 6.3 (1990), pp. 153–166. DOI: 10.1007/BF01911006.

[Mey+10] Quirin Meyer et al. “On Floating-Point Normal Vectors”. In: *Computer Graphics Forum* 29.4 (Aug. 26, 2010), pp. 1405–1409. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2010.01737.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2010.01737.x (visited on Feb. 3, 2019).

[Nah+11] Jae-Ho Nah et al. "T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing". In: ACM Press, 2011, p. 1. ISBN: 978-1-4503-0807-6. DOI: 10.1145/2024156.2024194. URL: http://dl.acm.org/citation.cfm?doid=2024156.2024194 (visited on Apr. 19, 2018).

[Nah+14] Jae-Ho Nah et al. "RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices". In: *ACM Transactions on Graphics* 33.5 (Sept. 23, 2014), pp. 1–15. ISSN: 07300301. DOI: 10.1145/2629634. URL: http://dl.acm.org/citation.cfm?doid=2672594.2629634 (visited on Apr. 17, 2018).

[Nav+07] Paul Arthur Navrátil et al. "Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization". In: IEEE, Sept. 2007, pp. 95–104. ISBN: 978-1-4244-1629-5. DOI: 10.1109/RT.2007.4342596. URL: http://ieeexplore.ieee.org/document/4342596/ (visited on May 2, 2018).

[NVIa] NVIDIA. *NVIDIA Turing GPU Architecture*. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on Dec. 4, 2018).

[NVIb] NVIDIA. *Parallel Thread Execution ISA Version 6.3, Data Movement and Conversion Instructions: Prefetch, Prefetchu*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-prefetch-prefetchu (visited on Jan. 24, 2019).

[nvi17] nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Aug. 2017, p. 58. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on July 6, 2021).

[Pha+97] Matt Pharr et al. "Rendering Complex Scenes with Memory-Coherent Ray Tracing". In: ACM Press, 1997, pp. 101–108. ISBN: 978-0-89791-896-1. DOI: 10.1145/258734.258791. URL: http://portal.acm.org/citation.cfm?doid=258734.258791 (visited on May 2, 2018).

[Res05] Alex Reshetov. "Multi-Level Ray Tracing Algorithm". In: *ACM Transactions on Graphics (TOG)* (2005), p. 10.

[RW80] Steven M Rubin and Turner Whitted. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes". In: *ACM SIGGRAPH Computer Graphics* (1980), p. 7.

[Sar18] Navid Saremi. "Low Level GPU Performance Characteristics Using Vendor Independent Benchmarks". Utrecht University, July 2018. 58 pp. URL: https://drive.google.com/file/d/15j6a_jKs-bNFCpvrrakLjq6t8Dx16j0A/view.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial Splits in Bounding Volume Hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (2009), pp. 7–14. ISSN: 9781605586038. DOI: 10.1145/1572769.1572771. URL: http://dl.acm.org/citation.cfm?id=1572771.

[SLa03] Jörg Schmittler, Alexander Leidinger, and et al. "A Virtual Memory Architecture For Real-Time Ray . . ." In: *Computer & Graphics 27, 5 (October), 693–699. Issn.* 2003, pp. 97–8493.

[Spj+09] J. Spjut et al. "TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.12 (Dec. 2009), pp. 1802–1815. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2009.2028981. URL: http://ieeexplore.ieee.org/document/5324031/ (visited on Apr. 23, 2018).

[SS92] Kelvin Sung and Peter Shirley. "Ray Tracing with the BSP Tree". In: *Graphics Gems III (IBM Version)*. Elsevier, 1992, pp. 271–274.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. "SaarCOR: A Hardware Architecture for Ray Tracing". In: *Conference on Graphics Hardware* (2002), pp. 1–11. ISSN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569051.

[Tsa09] John A. Tsakok. “Faster Incoherent Rays: Multi-BVH Ray Stream Tracing”. In: *Proceedings of the 1st ACM Conference on High Performance Graphics - HPG ’09*. The 1st ACM Conference. New Orleans, Louisiana: ACM Press, 2009, p. 151. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572793. URL: http://portal.acm.org/citation.cfm?doid=1572769.1572793 (visited on Nov. 4, 2018).

[VAS16] K Vaidyanathan, T Akenine-Möller, and M Salvi. “Watertight Ray Traversal with Reduced Precision”. In: (2016), p. 8.

[vdZRJ95] Maurice van der Zwaan, Erik Reinhard, and Frederik W. Jansen. “Pyramid Clipping for Efficient Ray Traversal”. In: *Rendering Techniques ’95*. Ed. by Patrick M. Hanrahan and Werner Purgathofer. Vienna: Springer Vienna, 1995, pp. 1–10. ISBN: 978-3-211-82733-8. DOI: 10.1007/978-3-7091-9430-0_1. URL: http://link.springer.com/10.1007/978-3-7091-9430-0_1 (visited on Nov. 4, 2018).

[Vii+16] Timo Viitanen et al. “Multi Bounding Volume Hierarchies for Ray Tracing Pipelines”. In: ACM Press, 2016, pp. 1–4. ISBN: 978-1-4503-4541-5. DOI: 10.1145/3005358.3005384. URL: http://dl.acm.org/citation.cfm?doid=3005358.3005384 (visited on Apr. 17, 2018).

[Wal+01] Ingo Wald et al. “Interactive Rendering with Coherent Ray Tracing”. In: *Computer Graphics Forum* 20.3 (2001), pp. 153–165. DOI: 10.1111/1467-8659.00508. URL: http://doi.wiley.com/10.1111/1467-8659.00508.

[Wal+05] I. Wald et al. “Faster Isosurface Ray Tracing Using Implicit KD-Trees”. In: *IEEE Transactions on Visualization and Computer Graphics* 11.5 (Sept. 2005), pp. 562–572. ISSN: 1077-2626. DOI: 10.1109/TVCG.2005.79. URL: http://ieeexplore.ieee.org/document/1471693/ (visited on Nov. 19, 2018).

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. “Getting Rid of Packets - Efficient SIMD Single-Ray Traversal Using Multi-Branching BVHs”. In: *RT’08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings*. 2008, pp. 49–57. DOI: 10.1109/RT.2008.4634620.

[WBS07] Ingo Wald, Solomon Boulous, and Peter Shirley. “Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies”. In: *ACM Transactions on Graphics* 26.1 (2007), p. 18.

[Whi79] Turner Whitted. “An Improved Illumination Model for Shaded Display”. In: *ACM SIGGRAPH Computer Graphics* 13.2 (1979), pp. 14–14. ISSN: 0897910044. DOI: 10.1145/965103.807419. URL: http://portal.acm.org/citation.cfm?doid=965103.807419.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. “B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes”. In: *Work* (June 2014 2006), pp. 67–77. ISSN: 3905673371. DOI: 10.1145/1283900.1283912. URL: http://portal.acm.org/citation.cfm?id=1283912.

[Woo06] Sven Woop. “DRPU A Programmable Hardware Architecture for Real-Time Ray Tracing of Coherent Dynamic Scenes”. Saarland University, 2006. URL: http://sven-woop.de/papers/2006-Thesis-DRPU-V1.1.pdf (visited on Apr. 23, 2018).

[WSS05] Sven Woop, Jorg Schmittler, and Philipp Slusallek. “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing”. In: (2005), p. 11.

[YKL17] Henri Ylitie, Tero Karras, and Samuli Laine. “Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs”. In: *Proceedings of High Performance Graphics on - HPG ’17*. High Performance Graphics. Los Angeles, California: ACM Press, 2017, pp. 1–13. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105773. URL: http://dl.acm.org/citation.cfm?doid=3105762.3105773 (visited on Nov. 28, 2018).

A. Structure Listings

Listing A.1: Ray Layout

```
struct Ray
{
    uint64 stack;
    uint depthQueueCount;
    uint extraIdx;

    // origin
    float4 o; // .w is float t (dist)

    // direction (not inv direction)
    float4 d; // .w is uint idx (ie tri idx)
};
```

Listing A.2: RayExtra Layout

```
struct RayExtra
{
    float3 E, T;

    // NEE specifics
    // if this is an NEE ray, this is the emissive
    // triangle we're trying to get to.
    uint neeTarget;

    float NdotL;

    uint flags;
    uint pixelIdx;
};
```

B. Tuning Parameters

Parameter	Description	Section	Valid Values
SCRWIDTH	Horizontal resolution of output image		
SCRHEIGHT	Vertical resolution of output image		
NUM_SPP	Number of Samples Per Pixel		
MIN_TREELETSIZE	Minimum size of a treelet		
TREELETSIZE	Maximum number of nodes in a BVH branch before creating a treelet		
BVH_MAXDEPTH	Max depth of a top-level BVH branch before creating a treelet		
BUCKET_ALGO	Controls selection algorithm for next bucket	3.2.3	MOSTFULL, RANDOM, NEXTBUCKET
DIRECT	DIRECT (copy by value) or INDIRECT (external array index) storage of rays in RayQueues	3.4.2	DIRECT or INDIRECT
MAXDEPTH	Maximum number of bounces for a ray before termination		
NEEMODE	Next Event Estimation		ON, OFF, HALFHALF
NEXTWORK	Algorithm to determine ordering of Ray creatiion		PIXELFIRST or FRAMEFIRST
NUM_WARPS	Active number of CUDA warps to invoke		
RAYCOUNT	Total number of ray slots to allocate		
REGCOUNT	Number of registers to allocate to kernels. Passed to CUDA's CU_JIT_MAX_REGISTERS parameter		
TOPLEVEL	Selects variant of top-level traversal algorithm to use	3.3.1	NAIVE, NAIVE_INTERLOCKED
ACTIVE_RAY_LOWATER	Number of active rays to trigger new ray generation		
ACTIVE_RAY_HIWATER	Number of active rays to cease new ray generation		
QUEUECOUNT	Number of RayQueue objects to allocate		
QUEUESIZE	Capacity of RayQueue object in Rays		
NUM_WARPS	Number of CUDA warps to launch		
SINGLE_LANE	Force only a single thread to be launched for debugging purposes		

Table B.1.: Compile-time parameters to the system

C. Initial-Run Results Table

Scene	NextWork	Direct	TreeletSize	RayCount	CrawlTime	ResToplevelNodeCount	ResTreeletCount
dragon-plane	FRAMEFIRST	DIRECT	800	10240	31.3916	95	71
dragon-plane	FRAMEFIRST	DIRECT	200	10240	31.5113	111	83
dragon-plane	FRAMEFIRST	DIRECT	1600	10240	31.9685	55	41
dragon-plane	FRAMEFIRST	DIRECT	2000	10240	32.4625	51	38
dragon-plane	PIXELFIRST	DIRECT	200	10240	35.6	111	83
dragon-plane	PIXELFIRST	DIRECT	800	10240	35.9678	95	71
dragon-plane	PIXELFIRST	DIRECT	1600	10240	38.5691	55	41
dragon-plane	PIXELFIRST	DIRECT	2000	10240	38.9551	51	38
dragon-plane	PIXELFIRST	DIRECT	200	1048576	40.6952	111	83
dragon-plane	PIXELFIRST	INDIRECT	2000	1024	47.178	51	38
dragon-plane	PIXELFIRST	INDIRECT	1600	1024	47.388	55	41
dragon-plane	PIXELFIRST	INDIRECT	200	1048576	47.9515	111	83
dragon-plane	PIXELFIRST	INDIRECT	800	1048576	48.5205	95	71
dragon-plane	PIXELFIRST	INDIRECT	1600	1048576	49.0762	55	41
dragon-plane	PIXELFIRST	INDIRECT	800	1024	49.0766	95	71
dragon-plane	PIXELFIRST	INDIRECT	2000	1048576	49.4686	51	38
dragon-plane	PIXELFIRST	INDIRECT	200	10240	49.6134	111	83
dragon-plane	PIXELFIRST	INDIRECT	200	1024	49.7819	111	83
dragon-plane	PIXELFIRST	INDIRECT	1600	10240	49.9643	55	41
dragon-plane	PIXELFIRST	INDIRECT	800	10240	50.0471	95	71
dragon-plane	PIXELFIRST	INDIRECT	2000	10240	50.4176	51	38
dragon-plane	PIXELFIRST	DIRECT	800	1048576	51.8846	95	71
dragon-plane	PIXELFIRST	DIRECT	1600	1048576	52.442	55	41
dragon-plane	PIXELFIRST	DIRECT	2000	1048576	52.573	51	38
dragon-plane	FRAMEFIRST	DIRECT	200	1048576	53.3317	111	83

dragon-plane	FRAMEFIRST	DIRECT	800	1048576	53.5255	95	71
dragon-plane	PIXELFIRST	DIRECT	1600	1024	54.1857	55	41
dragon-plane	PIXELFIRST	DIRECT	2000	1024	54.259	51	38
dragon-plane	FRAMEFIRST	DIRECT	1600	1048576	54.3778	55	41
dragon-plane	FRAMEFIRST	DIRECT	2000	1048576	54.4668	51	38
dragon-plane	FRAMEFIRST	DIRECT	2000	1024	56.3235	51	38
dragon-plane	FRAMEFIRST	DIRECT	1600	1024	56.3612	55	41
dragon-plane	PIXELFIRST	DIRECT	800	1024	57.1981	95	71
dragon-plane	PIXELFIRST	DIRECT	200	1024	58.0435	111	83
dragon-plane	FRAMEFIRST	DIRECT	800	1024	59.8121	95	71
dragon-plane	FRAMEFIRST	DIRECT	200	1024	61.0349	111	83
dragon-plane	FRAMEFIRST	INDIRECT	2000	1024	61.9878	51	38
dragon-plane	FRAMEFIRST	INDIRECT	1600	1024	62.3267	55	41
dragon-plane	FRAMEFIRST	INDIRECT	200	1048576	63.5158	111	83
dragon-plane	FRAMEFIRST	INDIRECT	800	1048576	63.7678	95	71
dragon-plane	FRAMEFIRST	INDIRECT	200	10240	63.7755	111	83
dragon-plane	FRAMEFIRST	INDIRECT	1600	10240	64.3306	55	41
dragon-plane	FRAMEFIRST	INDIRECT	1600	1048576	64.4131	55	41
dragon-plane	FRAMEFIRST	INDIRECT	800	10240	64.5181	95	71
dragon-plane	FRAMEFIRST	INDIRECT	2000	1048576	64.7786	51	38
dragon-plane	FRAMEFIRST	INDIRECT	2000	10240	64.9063	51	38
dragon-plane	FRAMEFIRST	INDIRECT	800	1024	64.9686	95	71
dragon-plane	FRAMEFIRST	INDIRECT	200	1024	66.0856	111	83
sponza	PIXELFIRST	INDIRECT	1600	1024	128.751	11	8
sponza	PIXELFIRST	INDIRECT	200	1024	128.763	11	8
sponza	PIXELFIRST	INDIRECT	800	1024	128.786	11	8
sponza	PIXELFIRST	INDIRECT	2000	1024	130.534	7	5
sponza	PIXELFIRST	INDIRECT	200	10240	137.864	11	8
sponza	PIXELFIRST	INDIRECT	800	10240	137.881	11	8
sponza	PIXELFIRST	INDIRECT	1600	10240	137.93	11	8
sponza	PIXELFIRST	INDIRECT	2000	10240	138.803	7	5
sponza	PIXELFIRST	INDIRECT	200	1048576	139.45	11	8
sponza	PIXELFIRST	INDIRECT	1600	1048576	139.464	11	8
sponza	PIXELFIRST	INDIRECT	800	1048576	139.469	11	8
sponza	PIXELFIRST	INDIRECT	2000	1048576	139.857	7	5

sponza	PIXELFIRST	DIRECT	1600	1024	151.418	11	8
sponza	PIXELFIRST	DIRECT	800	1024	151.419	11	8
sponza	PIXELFIRST	DIRECT	200	1024	151.497	11	8
sponza	PIXELFIRST	DIRECT	2000	1024	153.951	7	5
sponza	PIXELFIRST	DIRECT	200	1048576	158.681	11	8
sponza	PIXELFIRST	DIRECT	1600	10240	158.79	11	8
sponza	PIXELFIRST	DIRECT	800	1048576	158.821	11	8
sponza	PIXELFIRST	DIRECT	200	10240	158.854	11	8
sponza	PIXELFIRST	DIRECT	800	10240	158.856	11	8
sponza	PIXELFIRST	DIRECT	1600	1048576	159.002	11	8
sponza	PIXELFIRST	DIRECT	2000	1048576	159.362	7	5
sponza	PIXELFIRST	DIRECT	2000	10240	159.53	7	5
sponza	FRAMEFIRST	DIRECT	800	1024	159.66	11	8
sponza	FRAMEFIRST	DIRECT	1600	1024	159.668	11	8
sponza	FRAMEFIRST	DIRECT	200	1024	159.679	11	8
sponza	FRAMEFIRST	DIRECT	2000	1024	161.852	7	5
sponza	FRAMEFIRST	DIRECT	1600	1048576	163.488	11	8
sponza	FRAMEFIRST	DIRECT	200	1048576	163.655	11	8
sponza	FRAMEFIRST	DIRECT	800	10240	163.961	11	8
sponza	FRAMEFIRST	DIRECT	200	10240	164.116	11	8
sponza	FRAMEFIRST	DIRECT	1600	10240	164.153	11	8
sponza	FRAMEFIRST	DIRECT	800	1048576	164.177	11	8
sponza	FRAMEFIRST	DIRECT	2000	1048576	164.288	7	5
sponza	FRAMEFIRST	DIRECT	2000	10240	164.525	7	5
sponza	FRAMEFIRST	INDIRECT	200	1024	172.768	11	8
sponza	FRAMEFIRST	INDIRECT	1600	1024	172.78	11	8
sponza	FRAMEFIRST	INDIRECT	800	1024	172.805	11	8
sponza	FRAMEFIRST	INDIRECT	2000	1024	174.919	7	5
sponza	FRAMEFIRST	INDIRECT	800	10240	181.109	11	8
sponza	FRAMEFIRST	INDIRECT	200	10240	181.116	11	8
sponza	FRAMEFIRST	INDIRECT	1600	10240	181.138	11	8
sponza	FRAMEFIRST	INDIRECT	2000	10240	182.321	7	5
sponza	FRAMEFIRST	INDIRECT	800	1048576	182.731	11	8
sponza	FRAMEFIRST	INDIRECT	1600	1048576	182.745	11	8
sponza	FRAMEFIRST	INDIRECT	200	1048576	182.755	11	8

sponza	FRAMEFIRST	INDIRECT	2000	1048576	182.921	7	5

D. RayCount-Run Results Table

Scene	NextWork	Direct	TreeletSize	RayCount	CrawlTime	ResToplevelNodeCount	ResTreeletCount
dragon-plane	FRAMEFIRST	DIRECT	800	2097152	42.7255	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	1048576	42.2104	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	524288	42.224	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	262144	42.5673	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	131072	42.7765	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	65536	43.0972	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	32768	43.1299	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	16384	43.5578	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	8192	43.7934	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	4096	44.2371	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	2048	45.3382	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	1024	46.8913	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	512	51.4182	95	71
dragon-plane	FRAMEFIRST	DIRECT	800	256	59.6428	95	71
sponza	FRAMEFIRST	DIRECT	800	2097152	129.177	11	8
sponza	FRAMEFIRST	DIRECT	800	1048576	128.696	11	8
sponza	FRAMEFIRST	DIRECT	800	524288	128.686	11	8
sponza	FRAMEFIRST	DIRECT	800	262144	128.126	11	8
sponza	FRAMEFIRST	DIRECT	800	131072	128.452	11	8
sponza	FRAMEFIRST	DIRECT	800	65536	128.425	11	8
sponza	FRAMEFIRST	DIRECT	800	32768	128.705	11	8
sponza	FRAMEFIRST	DIRECT	800	16384	127.846	11	8
sponza	FRAMEFIRST	DIRECT	800	8192	128.879	11	8
sponza	FRAMEFIRST	DIRECT	800	4096	128.752	11	8
sponza	FRAMEFIRST	DIRECT	800	2048	127.668	11	8

sponza	FRAMEFIRST	DIRECT	800	1024	124.919	11	8
sponza	FRAMEFIRST	DIRECT	800	512	120.878	11	8
sponza	FRAMEFIRST	DIRECT	800	256	115.416	11	8

E. QueueCount-Run Results Table

For every run, treeletSize was 800 and rayCount 1048576.

Scene	Direct	QueueSize	CrawlTime	MemThrput	MemBusy	MaxBandwidth	L2HitRate	MemPipesBusy	L1HitRate
dragon-plane	DIRECT	2048	39.9134	247.76	0.05	0.06	76.28	0.03	66.74
dragon-plane	DIRECT	1024	39.7674	203.3	0.04	0.05	75.15	0.03	65.81
dragon-plane	DIRECT	512	39.9843	246.22	0.05	0.06	76.44	0.03	66.75
dragon-plane	DIRECT	256	40.0036	202.63	0.04	0.05	75.29	0.03	65.68
dragon-plane	DIRECT	128	41.2178	245.61	0.05	0.06	76.4	0.03	66.61
dragon-plane	DIRECT	64	42.2853	201.66	0.04	0.05	75.33	0.03	65.54
dragon-plane	DIRECT	32	44.5195	243.99	0.05	0.06	76.67	0.03	66.44
dragon-plane	INDIRECT	2048	47.2589	200.1	0.04	0.05	75.55	0.03	65.03
dragon-plane	INDIRECT	1024	47.9698	243.03	0.05	0.06	76.3	0.03	66.24
dragon-plane	INDIRECT	512	48.1684	199.51	0.04	0.05	75.48	0.03	64.96
dragon-plane	INDIRECT	256	48.6586	240.07	0.05	0.06	76.76	0.03	65.8
dragon-plane	INDIRECT	128	49.1721	197.64	0.04	0.05	75.31	0.03	64.7
dragon-plane	INDIRECT	64	50.3644	235.2	0.05	0.05	77.35	0.03	64.94
dragon-plane	INDIRECT	32	52.5423	195.79	0.04	0.05	75.62	0.03	63.92
sponza	DIRECT	2048	119.092	259.06	0.04	0.06	73.22	0.03	58.3
sponza	DIRECT	1024	119.904	220.25	0.04	0.05	72.81	0.03	58.24
sponza	DIRECT	512	120.749	258.17	0.04	0.06	73.05	0.03	58.23
sponza	DIRECT	256	121.686	219.92	0.04	0.05	72.8	0.03	58.17
sponza	DIRECT	128	124.996	256.95	0.04	0.06	73.39	0.03	58.15
sponza	DIRECT	64	128.67	219.38	0.04	0.05	73.11	0.03	58.08
sponza	DIRECT	32	133.046	256.07	0.04	0.06	73.35	0.03	58.01
sponza	INDIRECT	2048	133.516	218.84	0.04	0.05	72.69	0.03	57.91
sponza	INDIRECT	1024	135.232	255.26	0.04	0.06	73.47	0.03	57.8

sponza	INDIRECT	512	136.372	217.28	0.04	0.05	73.12	0.03	57.68
sponza	INDIRECT	256	137.858	252.86	0.04	0.06	73.56	0.03	57.49
sponza	INDIRECT	128	140.317	215.24	0.04	0.05	73.48	0.03	57.36
sponza	INDIRECT	64	143.967	248.19	0.04	0.06	73.93	0.03	56.92
sponza	INDIRECT	32	148.129	213.06	0.04	0.05	73.52	0.03	57.01