Hardware-Accelerated Dual-Split Trees

DAQI LIN, University of Utah
ELENA VASIOU, University of Utah
CEM YUKSEL, University of Utah
DANIEL KOPTA, University of Utah
ERIK BRUNVAND, University of Utah

Bounding volume hierarchies (BVH) are the most widely used acceleration structures for ray tracing due to their high construction and traversal performance. However, the bounding planes shared between parent and children bounding boxes is an inherent storage redundancy that limits further improvement in performance due to the memory cost of reading these redundant planes. Dual-split trees can create identical space partitioning as BVHs, but in a compact form using less memory by eliminating the redundancies of the BVH structure representation. This reduction in memory storage and data movement translates to faster ray traversal and better energy efficiency. Yet, the performance benefits of dual-split trees are undermined by the processing required to extract the necessary information from their compact representation. This involves bit manipulations and branching instructions which are inefficient in software. We introduce hardware acceleration for dual-split trees and show that the performance advantages over BVHs are emphasized in a hardware ray tracing context that can take advantage of such acceleration. We provide details on how the operations needed for decoding dual-split tree nodes can be implemented in hardware and present experiments in a number of scenes with different sizes using path tracing. In our experiments, we have observed up to 31% reduction in render time and 38% energy saving using dual-split trees as compared to binary BVHs representing identical space partitioning.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Graphics processors**; • **Computer systems organization** → *Parallel architectures.*

Additional Key Words and Phrases: acceleration structures

ACM Reference Format:
Daqi Lin, Elena Vasiou, Cem Yuksel, Daniel Kopta, and Erik Brunvand. 2020. Hardware-Accelerated Dual-Split Trees. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 20 (August 2020), 21 pages. https://doi.org/10.1145/3406185

1 INTRODUCTION

The bounding volume hierarchy (BVH) structure has been a popular space partitioning method for ray tracing. Yet, BVHs have inherent redundancies that inflate their storage cost. Considering that data movement can quickly become a bottleneck for modern, highly-parallel architectures, particularly for computing tasks like ray traversal, minimizing the storage overhead and data movement required for space partitioning is an important step for improving ray tracing performance and reducing its energy cost.

Recently, dual-split trees [Lin et al. 2019a] were introduced as a compact data structure that can represent identical space partitioning as BVHs but with significantly reduced storage cost. A dual-split tree can be quickly constructed from a given BVH, and its storage reduction was shown

Authors' addresses: Daqi Lin, University of Utah; Elena Vasiou, University of Utah; Cem Yuksel, University of Utah; Daniel Kopta, University of Utah; Erik Brunvand, University of Utah.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, https://doi.org/10.1145/3406185.

to provide faster software ray tracing performance on the CPU. Yet, the reported performance improvement of dual-split trees has been relatively minor in comparison to the substantial reduction they provide in storage. This is mainly because decoding dual-split tree nodes for extracting the space partitioning information involves branching and various bit manipulation operations, resulting in a considerable computation overhead for a software implementation.

In this paper, we introduce hardware acceleration for dual-split trees: specifically specialized hardware logic for decoding dual-split tree nodes. Since the decoding operations mainly involve branching and bit manipulation, they can be efficiently handled via hardware implementation, overcoming the computation overhead of dual-split trees. To test the performance improvement of our hardware decoder for dual-split trees, we provide results using a software path tracer running on TRaX [Spjut et al. 2009], a well-studied highly-parallel architecture designed for ray tracing that is supported by a detailed cycle-accurate simulator [Shkurko et al. 2018]. Our results show that our hardware acceleration for dual-split trees can achieve up to 31% reduction in render time and 38% energy saving as compared to using binary BVHs with hardware-accelerated ray-box intersections. We also include comparisons to BVHs with higher arity, showing that hardware-accelerated dual-split trees outperform all tested BVH variants in ray traversal performance, energy use, and storage.

2 RELATED WORK

Ray tracing in general is a memory bound task no matter what architecture is used, and that memory bottleneck is often the large scenes encoded in an acceleration structure such as a BVH.

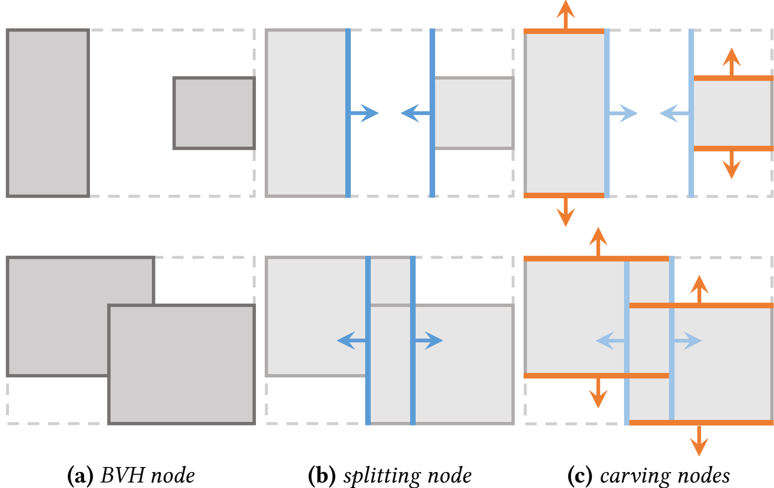
Therefore, reducing the memory footprint of scene traversal through the acceleration structure is a crucial part of increasing ray tracing performance. There have been numerous proposals to use specialized fixed-function hardware units for acceleration structure traversal [Kim et al. 2010a, 2012; Lee et al. 2013; Nah et al. 2014; Schmittler et al. 2002, 2004; Woop et al. 2006a, 2005]. While these have been shown to be effective at accelerating traversal, largely through more efficient memory access, they are still bound by the redundant encoding of the structures themselves, which limits their effectiveness.

2.1 BVH Optimizations

A recent trend is to reduce the storage requirement by either increasing the branching factor of the acceleration structure [Dammertz et al. 2008; Wald et al. 2008; Ylitie et al. 2017] or compressing the planes [Keely 2014; Lier et al. 2018; Selgrad et al. 2016; Ylitie et al. 2017]. When the arity of the tree increases, the number of intermediate nodes decreases and as a result fewer planes are shared between children and parent. A wider node (4 or 8 children) is attractive to current CPU or GPU architectures as computations on these wider nodes can benefit from SIMD computation. However, using wider nodes might not be an optimal solution in all cases. While compressing the planes either by directly or incrementally quantizing the bounding boxes can compress the storage to an arbitrary percentage, quantizing the bounds results in less tight bounding boxes, leading to more box and primitive intersections. In the case of incremental encoding, parent information must be pushed to the traversal stack, increasing memory traffic. Another line of work attempts to compress the pointer information in the nodes, by altering the tree structure to allocate fewer bits for storing child node or primitive addresses [Benthin et al. 2018; Kim et al. 2010b; Liktor and Vaidyanathan 2016].

2.2 Dual-Split Trees

Dual-split trees [Lin et al. 2019a] use a different way to remove the redundancy in a BVH. Instead of storing bounding boxes, each internal node stores two axis-aligned planes, with the function of

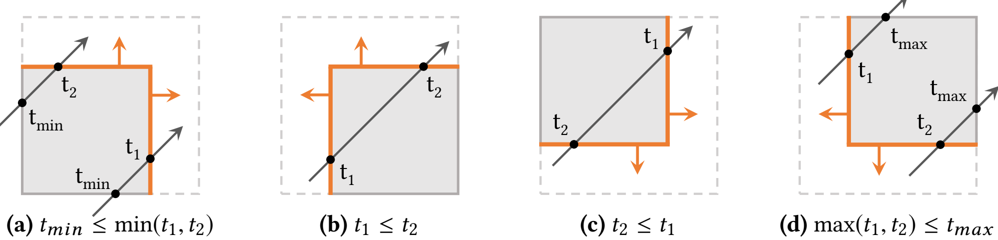


(a) BVH node (b) splitting node (c) carving nodes

Fig. 1. *Dual-split tree [Lin et al. 2019a] in 2D: (a) shows two configurations of a BVH node (child bounding boxes are shown in gray); (b) creates a splitting node (split planes depicted in blue); (c) adds carving nodes as children of the splitting node (carving planes depicted in orange). Each arrow indicates the plane normal, with the empty space on the positive side of the plane.*

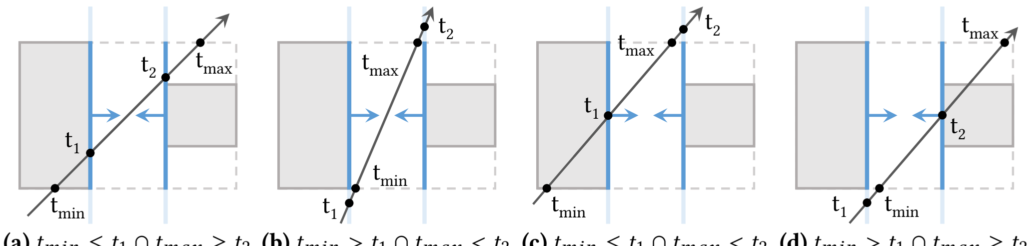
splitting the space into two (potentially overlapping) half-spaces or carving out the empty space in previously created half-spaces (Figure 1). As a result, they naturally avoid storing most shared planes between parent and child nodes. Dual-split trees encode the bounding planes implicitly by discarding the concept of the bounding box and using a hybrid of object and space partitioning as some previous methods (BIH [Wächter and Keller 2006], H-Tree [Havran et al. 2006], B-KD Tree [Woop et al. 2006b]), but have shown more memory footprint reduction and higher traversal performance on the CPU [Lin et al. 2019a]. While compact BVH [Fabianowski and Dingliana 2009] is an interesting alternative that also implicitly encodes the bounding planes, it generally requires more ray-plane intersections than the aforementioned hybrid methods [Lin et al. 2019a]. As a result, it potentially loads more data from memory despite having a size usually smaller than the hybrid methods. It is important to notice that dual-split trees and other hybrid methods per-se are not BVH compression methods as their definitions do not depend on hierarchical bounding boxes and can be built independently. But they can act as a way to losslessly compress the information stored in a BVH when converted from a BVH without changing the partitioning of objects and spaces.

Different from a kd tree that uses a single axis-aligned plane to split the child nodes of an internal node, a dual-split tree stores two axis-aligned planes per node and the planes can align to either the same or different axes. Each plane has a positive or negative 1D normal direction. The space along the normal direction is considered as empty in dual-split trees. This allows dual-split trees to define two types of internal nodes corresponding to two different ways of using the planes (Figure 1). In a *splitting node*, the planes split space into two halfs and they are known as *splitting planes*. In a *carving node*, the planes carve out empty space and they are called *carving planes*. Notice that the benefit of splitting the space using two planes is that the half-spaces can be overlapping or leaving a gap between them. While dual-split trees require the two splitting planes to align to the same axis, the carving planes can align to either the same axis to create a *single-axis carving node* (Figure 1c) or two different axes to create a *dual-axis carving node* (Figure 2). While a concept similar to the



(a) $t_{min} \leq \min(t_1, t_2)$ (b) $t_1 \leq t_2$ (c) $t_2 \leq t_1$ (d) $\max(t_1, t_2) \leq t_{max}$

Fig. 2. *Four configurations of carving planes (orange) in a dual-axis carving node [Lin et al. 2019a]. Subcaptions specify the conditions when a ray (black arrows) intersects the child within the node. These tests are similar to intersecting against a 2D axis-aligned bounding box.*



(a) $t_{min} \leq t_1 \cap t_{max} \geq t_2$ (b) $t_{min} > t_1 \cap t_{max} < t_2$ (c) $t_{min} \leq t_1 \cap t_{max} < t_2$ (d) $t_{min} > t_1 \cap t_{max} \geq t_2$

Fig. 3. *Four cases in splitting node traversal [Lin et al. 2019a]. Subcaptions specify the conditions when a ray (black arrows) intersects (a) both children, (b) neither child, (c) the closer child, and (d) the farther child. When a ray intersects the empty space (b), traversal stops.*

single-axis carving node is introduced in an augmented version of BIH [Wächter 2008], dual-split trees introduce dual-axis carving nodes, which allows using fewer nodes to carve the empty spaces.

Figure 3 shows an illustration of splitting node traversal. The intersection distances with the splitting planes that bound the near and far child nodes are denoted by t_1 and t_2, respectively. Similar to traversing a kd tree, each ray keeps track of a valid ray depth range $[t_{min}, t_{max}]$ and updates this range according to the ray-splitting plane intersection distances with which the range for the near child and the far child can be computed. When both children are intersected, the far child is pushed to the traversal stack. Notice that, different from kd trees, dual-split tree traversal continues after a hit is found (except for shadow rays), since the node bounds can overlap. In addition, internal dual-split tree nodes can encode empty space between the two splitting planes.

For a carving node, the ray's valid depth range is trimmed to the non-empty volume defined by the carving planes. If after trimming $t_{min} > t_{max}$, the ray traverses the empty space and misses the child of the node. A new node needs to be popped from the traversal stack just as in the case of missing a BVH bounding box. For a single-axis carving node, the ray intersects with the child only when both $t_{min} \leq t_1$ and $t_{max} \geq t_2$ are true. For a dual-axis carving node, there are four possible combinations of plane normal directions, creating four different "corner" cases as shown in Figure 2. The intersection resembles a 2D ray-box test, but with two bounds implicitly defined by the parent bounding box.

With a combination of splitting nodes and carving nodes, dual split trees largely eliminate the redundant storage of planes shared between parents and children, but additional meta-data need to be stored to indicate the reference axis and function of the planes. For example, in a dual-axis carving node, there are 3 different axes combination (xy, yz, xz) and 4 different "corner" types

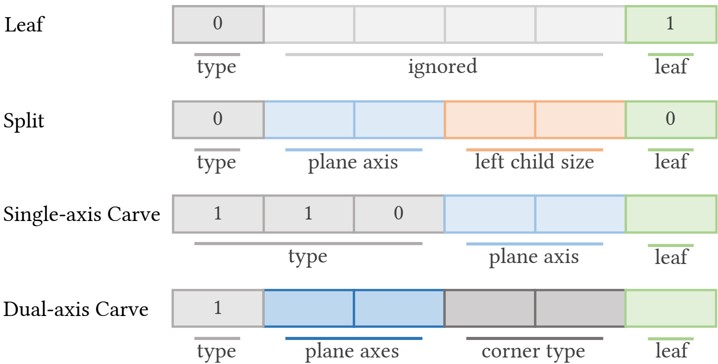


Fig. 4. *Node headers in our implementation of the dual-split tree consist of six bits allocated differently depending on the node type [Lin et al. 2019a].*

defined by the combination of plane normal directions, summing up to 12 different types of dual-axis carving nodes. While the information can be stored compactly in a 6-bit header [Lin et al. 2019a] (Figure 4), extracting the information and mapping it to different cases can be clumsy in software. A naive implementation might check the node header against all combinations of bits and use a switch statement to map it to different cases, which will be prohibitively expensive. A smarter implementation might combine many cases together and use bit twiddling tricks to make the code behave differently in different cases. Still, a software implementation needs to branch for splitting and carving nodes. However, using a specialized hardware pipeline decoding can be largely simplified and branching can be completely eliminated, which makes hardware implementation a very promising attempt to reduce the cost of dual-split tree traversal.

2.3 Ray Tracing Hardware

In the domain of custom hardware, there have been a variety of projects over the years that attempt to capture the intricacies of ray tracing and to exploit its inherent parallelism. In particular, many of these custom hardware systems have focused on single instruction multiple data (SIMD) parallelism similar to rasterizing GPUs [Schmittler et al. 2002, 2004; Woop et al. 2006a, 2005]. Other approaches such as StreamRay [Gribble and Ramani 2008; Ramani and Gribble 2009] provided a pre-filter operation to filter rays into SIMD-friendly groups for processing. These filtered sets of rays look like streams once they are assembled, but are not predictable in advance to allow for effective prefetching. With some additional overhead, SPMD architectures have been studied to take advantage of the parallel nature of ray tracing processing [Govindaraju et al. 2008; Kelm et al. 2009; Kopta et al. 2013, 2015, 2010; Shkurko et al. 2017; Spjut et al. 2009, 2008; Vasiou et al. 2019]. These systems assume a fairly standard cache/DRAM hardware architecture and rely on the behavior of the ray and scene data requests to leverage the memory system efficiently.

We use the previously developed TRaX architecture [Spjut et al. 2009] as the hardware ray tracing foundation for our tests. This architecture is in many ways the most generic of hardware ray tracing architectures as it does not utilize any fixed function rendering pipelines or other specialized logic units. The path tracing algorithm used in our tests is written in C++ without assuming any special purpose support, other than the hardware acceleration we are proposing.

The basic TRaX architecture, shown in Figure 5, consists of a large set of parallel computation resources that each have their own program counter. They operate based on a Single Program Multiple Data (SPMD) approach where all the parallel processing elements are running the same

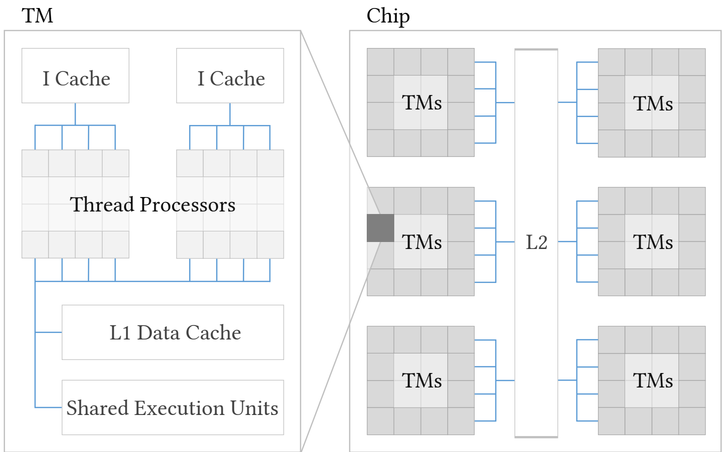


Fig. 5. *An overview of the TRaX architecture [Spjut et al. 2009].*

program (e.g. path tracing), but can be at different points in the program. This allows the processors to effectively share multi-banked cache and execution unit resources.

The fundamental unit of a TRaX-style architecture is a Thread Processor (TP). This is a small single-threaded single-issue processor that has its own program counter, register file, and basic datapath functional units (integer ALU, floating point add/sub/mult). These TPs are collected into a Thread Multiprocessor (TM) where multiple TPs share multi-banked L1 caches and more complex execution units. At the TM level, the shared execution units can be individual units such as square root or inverse, and can also be collected into multi-stage pipelines to efficiently compute application-specific operations such as ray-box or ray-triangle intersection [Kopta et al. 2013]. This run-time configuration of a computation pipeline is what we leverage to accelerate dual-split trees. TMs are collected together onto a chip and share multi-banked L2 caches that are backed by multiple DRAM channels. All of the specific parameters such as number of TPs in a TM, size and number of banks in caches, allocation of execution units, number of TMs on a chip, etc. are configurable and may change depending on the demands of a specific application.

TRaX is evaluated on a cycle-accurate simulator [Shkurko et al. 2018] that supports many thousands of processing elements working together in SPMD mode, and allows easy exploration of the effects of our proposed additional hardware. Because of the general nature of the TRaX architecture, and the completely software-controlled ray tracing used for evaluation, we are confident that we are reporting gains due to the dual-split tree acceleration that will be realizable across a wide variety of other types of hardware architectures for ray tracing.

Another domain of study in ray tracing hardware focuses on hardware acceleration structure builders. Hardware-accelerated kd tree builders have been proposed to reduce the kd tree construction cost with binned SAH or Morton code sorting [Liu et al. 2015; Nah et al. 2014]. Hardware BVH builders received more attention, since construction and update of BVHs are faster than kd trees. One example is the hardware microarchitecture for building binned SAH BVHs [Doyle et al. 2013, 2017]. More recently, hardware BVH builders including MergeTree [Viitanen et al. 2017] and PLOCTree [Viitanen et al. 2018] have taken advantage of Morton code sorting to substantially improve the hardware construction speed.

A given BVH can be quickly converted to a dual-split tree. An unoptimized software dual-split tree converter has been shown to take 38%-57% of the construction time of a highly optimized

software BVH builder [Lin et al. 2019a]. It is certainly possible to directly construct a dual-split tree without first building a BVH. In this paper, we concentrate on the ray traversal problem and leave hardware-accelerated dual-split tree construction to future work.

3 HARDWARE DUAL-SPLIT INTERSECTION PIPELINE

Dual-split trees [Lin et al. 2019a] substantially reduce the number of ray-plane intersections and the acceleration structure size compared to traditional BVHs. However, the performance gain is limited by a significantly more complex traversal kernel (see the supplemental document of Lin et al. [2019b]). First, decoding the node information, including node type and splitting/carving axis, requires many bit operations which add a non-trivial computational cost that undermines performance. Secondly, the large number of control branches for different node types and intersection conditions further affects performance, especially on general purpose GPU kernels. However, decoding and branching, while expensive in a software implementation, can be elegantly mapped to hardware with the cost largely eliminated. Therefore, we propose a hardware dual-split intersection pipeline that solves this key part of the dual-split tree traversal.

For the proposed hardware pipeline, we assume the node structure and tree layout proposed in Lin et al. [2019a]. A dual-split tree node starts with a packed word that we call *header-offset*, consisting of a 6-bit header (Figure 4) and 26-bit integer offset. The integer offset points to the left child in memory. The address of the right child is simply the left child address plus the left child size, since the tree is stored linearly in memory in a depth-first order with two child nodes always stored next to each other. If the node is an internal node, it has two other words that store either two splitting planes or carving planes as single-precision floats. Given a header-offset and two planes, and current ray information specified by *ray invdir* (the inverse of direction), *ray origin*, t_{min}, t_{max} (valid range of the ray), the pipeline can be seen as a black box that updates the ray information and produces the next step of the traversal. The next step is indicated by a 2-bit return value, with 0 being no intersection where the traversal stack is to be popped, 1 being single child intersection, 2 being intersecting both children, 3 being returning as a leaf node. This return code tells the program to change its control flow. When child intersection happens, the relative address of the child is returned as *offset* (next node to traverse) or $offset_{stack}$ (child node to be pushed to the stack), which is used to fetch the node for the next or later dual-split intersection. For a leaf node the offset points to the beginning of a range of its triangles in the global triangle index list (the last triangle index has the sign bit marked with 1).

An overview of the pipeline is provided in Figure 6 which shows the input/output and the data flow. As can be seen, dual-split intersection pipeline has 2 floating-point addition/subtraction (FPADD) and 2 floating-point multiplication (FPMUL) units, which are used for ray-plane intersection. Beyond that, there is just a little additional complexity for choosing the inputs for the ray-plane intersection and computing the new states (ray range, offset, return value) for traversal. The details about these parts (*ray component and plane selection logic*, *plane comparison and return value logic*, and *offset computation logic*) are in the appendix for the interested readers (Appendix A, B, and C). Note that compared to the floating point units, the latency, area, and power consumption of these additional logic blocks are relatively small. In comparison, a hardware ray-box intersection pipeline [Kopta 2016] with the same latency as the proposed dual-split intersection pipeline needs 6 FPADD and 6 FPMUL units, which costs substantially more area and energy (Table 2).

Notice that the hardware dual-split intersection pipeline completely eliminates the branching cost to test a ray with dual-split nodes since all the node types are treated uniformly, by taking advantage of the fact that all internal node types share the same intersection query between the ray and two planes, which only differs in which ray/plane components to choose and how to interpret the results. By merging the shared logic path using combinational logic, parallelizing the different

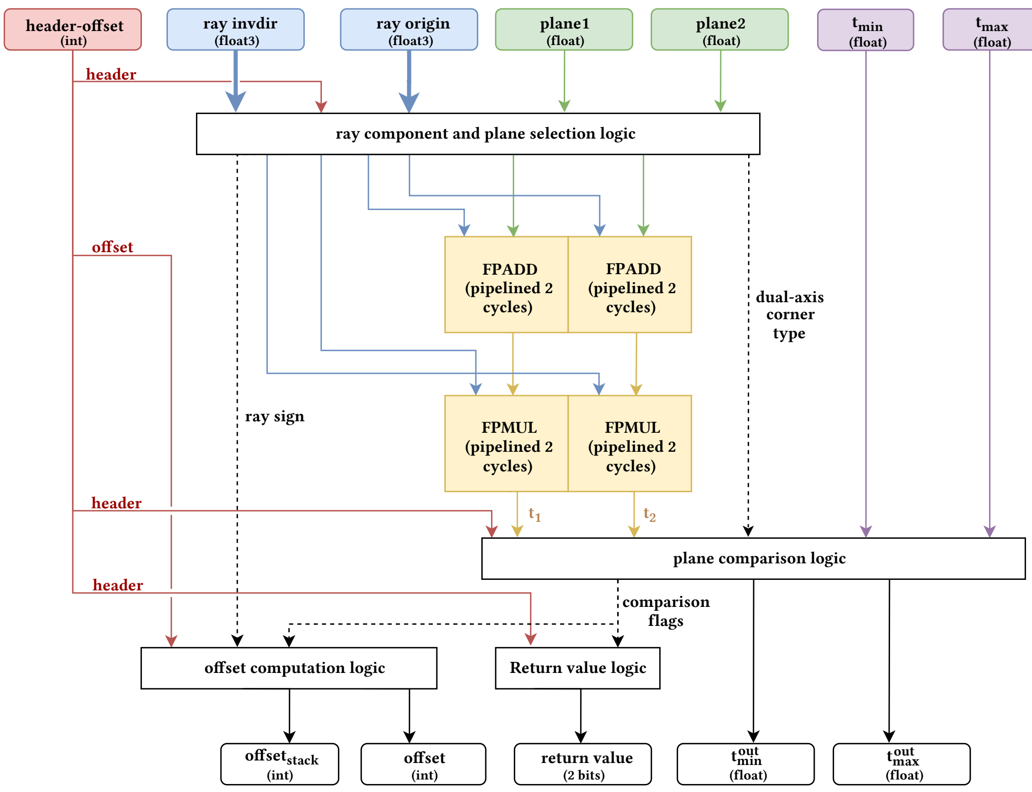


Fig. 6. *An overview of the hardware dual-split intersection pipeline. The FPADD units are used for subtractions. Arrows represent 32-bit data type (int, float) by default. Vector data (e.g. float3) are marked with thickened lines. Data under 2 bits are represented by dashed lines. Colored round boxes represent input registers. Uncolored round boxes represent outputs. Squared boxes stand for functional units. Unit areas are not to scale.*

logic paths (e.g. extracting corner type in dual-axis carving nodes), and choosing the result using multiplexers, we can use a single hardware pipeline for all node types, instead of using multiple hardware pipelines that are specific to node types or using software node testing, both of which require branching.

4 IMPLEMENTATION AND RESULTS

We choose TRaX [Spjut et al. 2009], a highly parallel and general MIMD architecture for high-performance ray tracing (described in Section 2.3), and the SimTRaX simulator [Shkurko et al. 2018] to simulate and evaluate our method, which includes a detailed DRAM model for accurate memory simulation [Chatterjee et al. 2012]. The general TRaX architecture permits many variations in terms of the basic configuration of computation and memory resources. The configuration used for this study is shown in Table 1. The simulated chip has a single large L2 cache of 4MB size and 128 banks to improve cache sharing. We connect 64 TMs to this L2 cache. Each TM consists of 32 TPs for a total of 2048 thread processing cores, has one L1 data cache sized at 64KB with 16 banks, and two instruction caches. We estimate the energy of the on-chip caches using Cacti 7 [Balasubramonian

Table 1. *Configuration of the simulated TRaX chip.*

Common System Parameters	
Technology Node	65nm CMOS
Clock Rate	1GHz
DRAM Memory	2GB GDDR5
Total Threads	2048
On-Chip Memory	
L2 Cache	4MB, 128 banks
Number of L2s	1
TMs / L2	64
TM Configuration	
TPs / TM	32
L1 Cache	64KB, 16 banks

et al. 2017]. The main memory of the system is simulated as GDDR5 DRAM with 16 32-bit channels, with a 512 GB/s maximum bandwidth and an effective clock rate of 8 GHz.

On the same architecture, we compare the ray tracing performance of our hardware-accelerated dual-split trees (DST) to bounding volume hierarchies with branching factors of two, four, and eight. We refer to them as BVH2, BVH4, and BVH8 in the following text. They use a single hardware-accelerated ray-box intersection pipeline [Kopta et al. 2013] which accepts a ray and an AABB, and outputs the hit result (true/false) and hit distance (for sorting the child nodes and detecting early termination). There is one ray-box pipeline per TM, so ray-box intersection tests are executed serially for each box in a BVH node. Similarly, we use a single dual-split pipeline per TM.

To experiment with reduced ray-box computational overhead, we also include tests with extended ray-box intersection pipelines that allow testing all boxes in a BVH node in parallel. Obviously, these extended pipelines have significantly more area and energy costs (by a factor of BVH arity) and the numbers of required units per TM are increased accordingly. Thus, these extended pipelines pack significantly more compute units. These BVHs with extended parallel ray-box pipelines are referred to as BVH2$^+$, BVH4$^+$, and BVH8$^+$ in the following text.

In addition, we compare to a dual-split tree with software decoding (DST*) to demonstrate the improvement when using hardware acceleration.

4.1 Hardware Pipeline Energy and Area Cost

The functional units which make up the pipelines are synthesized by Synopsys Design Compiler using a 65nm CMOS library, which allows us to calculate the area and energy used by each pipeline. The shared execution units using these functional units are configurable and can be used as individual functional units, or configured in a multi-stage pipeline to execute ray-box, ray-triangle [Kopta et al. 2013], or dual-split intersection. The overhead is low for these reconfigurable pipelines because existing functional units are temporarily reconfigured at run time using a combination of multiplexers and latches that already exist in the shared execution unit.

For the dual-split intersection pipeline additional logic gates are added to enable the required combinational bit operations required for this intersection (see Appendix A, B, and C for details). Because these are simple bit operations, this adds negligible overhead to the chip area. As Table 2 shows, the dual-split pipeline uses 61% less area than the box pipeline and consumes 56% less energy per access, while having the same latency in terms of clock cycles, assuming a 1GHz processor. A dual-split pipeline uses more bitwise units, latches, multiplexers than the box pipeline, but it requires fewer floating point arithmetic units which take up the majority of area and energy. In

Table 2. *Unit comparison between the hardware ray-box pipeline [Kopta 2016] and dual-split pipeline.*

		Ray-box	**Dual-split**
Integer Adder	(INTADD)	–	1 unit
Floating-point Adder	(FPADD)	6 units	2 units
Floating-point Multiplier	(FPMUL)	6 units	2 units
Floating-point Min/Max	(FPMIN/MAX)	12 units	3 units
Floating-point Comparison	(FPCMP)	2 units	4 units
	Total Area	0.1278 mm^2	0.0498 mm^2
	Energy per Operation	0.1377 nJ	0.0610 nJ
	Latency	8 cycles	8 cycles

particular, the number of both floating point addition/subtraction (FPADD) and floating point multiplication (FPMUL) units are reduced from six to two.

Note that, while a single dual-split pipeline has significantly less energy and area cost than a single ray-box pipeline, our tests with BVH2$^+$, BVH4$^+$, and BVH8$^+$ have energy and area costs of 2×, 4×, and 8× of single ray-box pipeline, respectively. Though, most of the energy cost in our tests stem from data movement and compute energy accounts for a relatively small percentage, the additional area costs for these extended pipelines are significant.

4.2 Acceleration Structure Construction

All acceleration structures being compared are converted from the same original binary BVH built using the Embree 3 BVH builder [Wald et al. 2014] with high quality settings and without triangle splitting. Thus, all acceleration structures present identical spatial partitioning. The binary BVH (BVH2) is used as our baseline in the results. The BVH4 is built by collapsing the binary BVH using the classical method of removing all odd levels [Dammertz et al. 2008]. For the BVH8, to avoid generating excessive empty children, we use the heuristic developed by Ylitie et al. [2017] to adaptively collapse the BVH2 to minimize tree size as well as reduce the traversal cost.

For the dual-split tree, we use the SAH-based converter described in Lin et al. [2019a], but adjust the cost of carving node intersection relative to triangle intersection. In the original CPU version of dual-split tree, the dual-axis carving node has a higher cost than the single-axis carving node. However, in our hardware-accelerated dual-split tree, all node types have the same cost since the latency of the dual-split intersection test does not depend on the node type. We build the dual-split tree using identical bounds as the source BVH, i.e., empty space is always carved before creating a splitting node.

For all acceleration structures, the tree is stored in linear depth-first order with child nodes belonging to the same parent always stored consecutively. An internal node of the BVH2 consists of two child bounding boxes stored using 12 single-precision floats, and a 4-byte child offset, with the child node types (leaf or internal) stored as the highest two bits sharing the same word the child offset, making each internal node 52 bytes long. We find this layout generally provides better performance than other layouts that store the same information in the chosen hardware architecture. A similar node layout is used for BVH4, which stores four child bounding boxes that use 24 floats and one word with packed offset and child node type, summing up to 100 bytes per node. For BVH8, the child node types are stored in an additional word since it will not leave enough space for the child offset if packed together. As a result, each BVH8 node has 200 bytes (192 bytes for child bounding boxes, 4 bytes for offset, and 4 bytes for node types). BVH4 and BVH8 nodes store degenerate bounding boxes for empty children.

Table 3. *Acceleration structure sizes for our test scenes. BVH2 is used as the baseline (100%).*

	Crytek Sponza 262K triangles		**Dragon Box** 870K triangles		**Vegetation** 1.1M triangles		**Hairball** 2.9M triangles		**Dragon Sponza** 6.6M triangles		**San Miguel** 10.5M triangles	
	Acceleration Structure Size (MiB)											
BVH2	7.03 MiB	**100%**	25.71 MiB	**100%**	22.74 MiB	**100%**	43.82 MiB	**100%**	181.61 MiB	**100%**	265.90 MiB	**100%**
BVH4	6.80 MiB	**97%**	24.77 MiB	**96%**	21.90 MiB	**96%**	42.24 MiB	**96%**	175.17 MiB	**96%**	256.33 MiB	**96%**
BVH8	5.64 MiB	**80%**	21.42 MiB	**83%**	19.33 MiB	**85%**	35.77 MiB	**82%**	150.53 MiB	**83%**	212.90 MiB	**80%**
DST	4.55 MiB	**65%**	15.35 MiB	**60%**	15.81 MiB	**70%**	28.91 MiB	**66%**	111.65 MiB	**61%**	175.01 MiB	**66%**

We test six different scenes with different levels of complexity listed in Table 3. The sizes of acceleration structures in different scenes are listed in the table. Notice that dual-split trees form by far the smallest acceleration structures for all scenes.

4.3 Ray Casting and Shadow Rays

We present performance results using ray casting and direct illumination with shadow rays in Table 4 in terms of render time, energy consumption, and number of cache lines transferred from memory. These render times and energy values are also visualized in Figure 7. Each generated frame is rendered at 1024×1024 resolution with one sample per pixel.

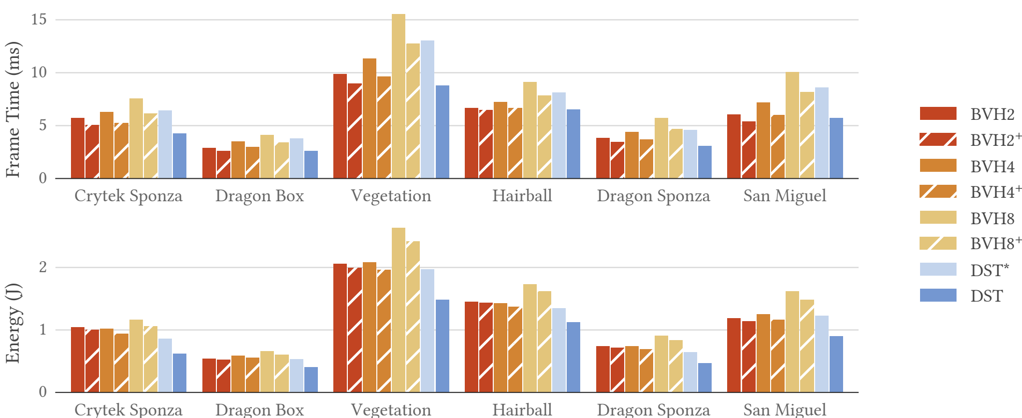
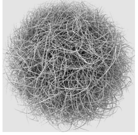


Fig. 7. *Ray casting and shadow rays performance: (top) render times and (bottom) energy use.*

Notice that hardware-accelerated dual-split trees consistently perform faster than BVH2, BVH4, and BVH8. Our hardware-accelerated dual-split trees reach render time reductions up to 25%, but this reduction can be as modest as 2% in some scenes. The only configuration that could perform faster rendering in some scenes (Dragon Box, Hairball, and San Miguel) is BVH2$^+$, which uses two parallel ray-box intersections. Note that each TM in our BVH2$^+$ tests needs 6 times the floating point adders and multipliers, as compared to the ones used for hardware-accelerated dual-split trees. Still, BVH2$^+$ trails behind hardware-accelerated dual-split tree in other scenes (Crytek Sponza, Vegetation, and Dragon Sponza).

Software implementation of dual-split trees (DST*), however, consistently perform slower than BVH2. We have observed up to 43% increase in render time with software dual-split trees, as

Table 4. *Ray casting and shadow rays (1 primary ray and 1 shadow ray per pixel) performance.*

												
	Crytek Sponza 262K triangles		**Dragon Box** 870K triangles		**Vegetation** 1.1M triangles		**Hairball** 2.9M triangles		**Dragon Sponza** 6.6M triangles		**San Miguel** 10.5M triangles	
Frame Time (milliseconds)												
BVH2	5.72 ms	**100%**	2.91 ms	**100%**	9.90 ms	**100%**	6.69 ms	**100%**	3.85 ms	**100%**	6.05 ms	**100%**
BVH2⁺	5.06 ms	**89%**	2.59 ms	**89%**	8.98 ms	**91%**	6.46 ms	**97%**	3.43 ms	**89%**	5.39 ms	**89%**
BVH4	6.28 ms	**110%**	3.52 ms	**121%**	11.37 ms	**115%**	7.26 ms	**109%**	4.39 ms	**114%**	7.18 ms	**119%**
BVH4⁺	5.23 ms	**91%**	2.95 ms	**102%**	9.64 ms	**97%**	6.67 ms	**100%**	3.67 ms	**95%**	6.00 ms	**99%**
BVH8	7.56 ms	**132%**	4.11 ms	**141%**	15.53 ms	**157%**	9.14 ms	**137%**	5.74 ms	**149%**	10.06 ms	**166%**
BVH8⁺	6.15 ms	**108%**	3.39 ms	**117%**	12.73 ms	**129%**	7.83 ms	**117%**	4.68 ms	**122%**	8.18 ms	**135%**
DST*	6.43 ms	**113%**	3.80 ms	**131%**	13.07 ms	**132%**	8.13 ms	**122%**	4.62 ms	**120%**	8.63 ms	**143%**
DST	4.27 ms	**75%**	2.62 ms	**90%**	8.81 ms	**89%**	6.52 ms	**98%**	3.08 ms	**80%**	5.74 ms	**95%**
Energy Consumption (J)												
BVH2	1.04 J	**100%**	0.54 J	**100%**	2.06 J	**100%**	1.45 J	**100%**	0.74 J	**100%**	1.19 J	**100%**
BVH2⁺	1.00 J	**96%**	0.52 J	**96%**	1.99 J	**97%**	1.43 J	**99%**	0.71 J	**96%**	1.14 J	**96%**
BVH4	1.02 J	**98%**	0.59 J	**109%**	2.08 J	**101%**	1.43 J	**99%**	0.74 J	**100%**	1.25 J	**105%**
BVH4⁺	0.94 J	**90%**	0.55 J	**102%**	1.96 J	**95%**	1.37 J	**94%**	0.69 J	**93%**	1.16 J	**97%**
BVH8	1.16 J	**112%**	0.66 J	**122%**	2.63 J	**128%**	1.73 J	**119%**	0.91 J	**123%**	1.62 J	**136%**
BVH8⁺	1.06 J	**102%**	0.60 J	**111%**	2.42 J	**117%**	1.62 J	**112%**	0.83 J	**112%**	1.48 J	**124%**
DST*	0.86 J	**82%**	0.53 J	**99%**	1.97 J	**96%**	1.35 J	**93%**	0.64 J	**86%**	1.23 J	**103%**
DST	0.62 J	**59%**	0.40 J	**75%**	1.48 J	**72%**	1.12 J	**77%**	0.47 J	**63%**	0.90 J	**76%**
# (Millions) Lines Transferred From Memory												
BVH2	2.18 M	**100%**	2.12 M	**100%**	5.07 M	**100%**	6.91 M	**100%**	2.81 M	**100%**	3.48 M	**100%**
BVH2⁺	2.16 M	**99%**	2.05 M	**97%**	5.09 M	**100%**	6.99 M	**101%**	2.77 M	**99%**	3.43 M	**99%**
BVH4	2.14 M	**98%**	2.19 M	**103%**	4.63 M	**91%**	5.91 M	**86%**	2.75 M	**98%**	3.40 M	**98%**
BVH4⁺	2.08 M	**95%**	2.10 M	**99%**	4.61 M	**91%**	6.00 M	**87%**	2.67 M	**95%**	3.32 M	**95%**
BVH8	2.21 M	**101%**	2.33 M	**110%**	4.47 M	**88%**	6.83 M	**99%**	2.85 M	**101%**	3.68 M	**106%**
BVH8⁺	2.17 M	**100%**	2.25 M	**106%**	4.48 M	**88%**	6.97 M	**101%**	2.79 M	**99%**	3.64 M	**105%**
DST*	2.10 M	**96%**	2.33 M	**110%**	4.03 M	**79%**	5.04 M	**73%**	2.45 M	**87%**	3.32 M	**95%**
DST	1.93 M	**89%**	2.08 M	**98%**	3.93 M	**78%**	5.11 M	**74%**	2.22 M	**79%**	3.11 M	**89%**

⁺ *BVH with extended parallel ray-box pipelines*
* *Dual-split tree with software decoding*

compared to BVH2. This shows the substantial impact of hardware acceleration for processing dual-split tree nodes.

In terms of energy consumption, hardware-accelerated dual-split trees provide unmatched performance, achieving 24% to 41% reduction, as compared to BVH2. The performance of hardware acceleration for dual-split trees also improves the cache performance and leads to memory traffic reduction, as compared to its software implementation.

4.4 Path Tracing

Path tracing with 5 bounces (without Russian Roulette) produces highly incoherent rays to stress the acceleration structure. We report our path tracing simulation results in Table 5 and Figure 8. In this case, we see a 21% to 31% reduction in render time with our hardware-accelerated dual-split trees over the baseline BVH2 across all test scenes. Our hardware-accelerated dual-split trees perform consistently faster than all other acceleration structures we tested, including ones that pack significantly more compute power with extended parallel ray-box pipelines.

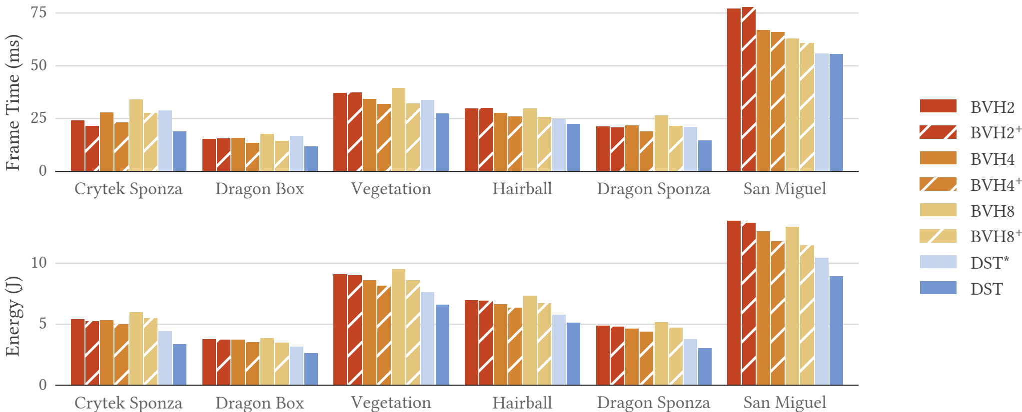


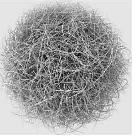
Fig. 8. *Path tracing performance: (top) render times and (bottom) energy use.*

Notice that the speedup of dual-split trees becomes more substantial when more memory needs to be transferred to the chip, which roughly grows with scene size. In San Miguel, our largest test scene, our method has a 28% reduction in render time over BVH2, while in the smallest scene, Crytek Sponza, the saving is only 21%. On the other hand, dual-split trees have better memory saving in both tree size and traversal memory traffic for scenes with more regular geometry (e.g. Crytek Sponza and Dragon Sponza, where a large number of objects are axis-aligned), because less carving nodes are generated [Lin et al. 2019a]. The hardware dual-split tree achieves the highest speedup of 1.46× in Dragon Sponza, which has the second highest triangle count and second highest tree size reduction ratio compared to BVH2 (Table 3).

Hardware-accelerated dual-split trees also reduce the energy consumption by 26%-38%, far better than the other acceleration structures. Software dual-split trees provide the second best performance in terms of energy consumption by a clear margin. These can be attributed to the fact that dual-split trees form significantly smaller acceleration structures. The improved computation speed provided by the hardware dual-split tree decoding also results in reduced energy consumption, as compared to software dual-split trees (DST*).

This suggests that the efficiency of our method is mainly correlated to the reduced memory traffic. Overall, our method reduces 29%-49% of the memory traffic of the baseline, measured in number of cache lines transferred from DRAM. However, without the hardware acceleration, the compute cost can limit the amount of speedup our method can achieve, as shown by the results of software-decoded dual-split tree. In high depth complexity scenes like San Miguel, the performance of dual-split trees is less effected by using a software decoder. However, in the other scenes (Crytek Sponza, Dragon Box, and Dragon Sponza), software-decoded dual-split trees are about 30% slower than hardware-decoded dual-split trees. In Crytek Sponza and Dragon Box, software-decoded dual split trees are even slower than BVH2. The reason is that computation time takes up a significant portion of the overall execution time, and the saving in memory cost is not substantial enough to offset that additional cost. Further, a software-decoded dual-split tree uses 12% to 32% more energy than hardware dual split trees, most of which is due to on-chip memory energy spent on fetching from instruction caches and register files during software decoding. Interestingly, compute energy itself does not differ much from hardware dual-split trees (Table 6). This is understandable since the hardware pipeline can utilize the same amount of functional units as in the case of software testing, despite consuming much less latency.

Table 5. *Path tracing (with 5 diffuse bounces) performance.*

	Crytek Sponza 262K triangles		**Dragon Box** 870K triangles		**Vegetation** 1.1M triangles		 **Hairball** 2.9M triangles		**Dragon Sponza** 6.6M triangles		**San Miguel** 10.5M triangles	
Frame Time (milliseconds)												
BVH2	24.1 ms	**100%**	15.3 ms	**100%**	37.1 ms	**100%**	29.8 ms	**100%**	21.3 ms	**100%**	77.0 ms	**100%**
BVH2$^+$	21.4 ms	**89%**	15.5 ms	**101%**	37.3 ms	**101%**	29.9 ms	**100%**	20.7 ms	**98%**	77.8 ms	**101%**
BVH4	27.8 ms	**115%**	15.9 ms	**104%**	34.3 ms	**92%**	27.6 ms	**92%**	21.7 ms	**102%**	67.0 ms	**87%**
BVH4$^+$	23.1 ms	**96%**	13.5 ms	**88%**	31.8 ms	**86%**	25.9 ms	**87%**	18.9 ms	**89%**	65.9 ms	**86%**
BVH8	34.1 ms	**141%**	17.7 ms	**116%**	39.5 ms	**107%**	29.8 ms	**100%**	26.5 ms	**125%**	62.9 ms	**82%**
BVH8$^+$	27.6 ms	**115%**	14.4 ms	**94%**	32.2 ms	**87%**	25.7 ms	**86%**	21.5 ms	**101%**	60.8 ms	**79%**
DST*	28.8 ms	**119%**	16.7 ms	**109%**	33.7 ms	**91%**	24.9 ms	**83%**	21.0 ms	**99%**	55.7 ms	**72%**
DST	19.0 ms	**79%**	11.8 ms	**77%**	27.5 ms	**74%**	22.4 ms	**75%**	14.6 ms	**69%**	55.6 ms	**72%**
Energy Consumption (J)												
BVH2	5.41 J	**100%**	3.78 J	**100%**	9.12 J	**100%**	6.97 J	**100%**	4.90 J	**100%**	13.46 J	**100%**
BVH2$^+$	5.23 J	**97%**	3.73 J	**99%**	9.01 J	**99%**	6.90 J	**99%**	4.78 J	**98%**	13.29 J	**99%**
BVH4	5.34 J	**99%**	3.73 J	**99%**	8.61 J	**94%**	6.64 J	**95%**	4.66 J	**95%**	12.61 J	**94%**
BVH4$^+$	5.00 J	**92%**	3.50 J	**93%**	8.13 J	**89%**	6.36 J	**91%**	4.36 J	**89%**	11.77 J	**87%**
BVH8	5.98 J	**111%**	3.87 J	**102%**	9.52 J	**104%**	7.35 J	**105%**	5.19 J	**106%**	12.97 J	**96%**
BVH8$^+$	5.49 J	**101%**	3.46 J	**92%**	8.58 J	**94%**	6.72 J	**96%**	4.69 J	**96%**	11.46 J	**85%**
DST*	4.44 J	**82%**	3.18 J	**84%**	7.61 J	**83%**	5.79 J	**83%**	3.79 J	**77%**	10.47 J	**78%**
DST	3.37 J	**62%**	2.65 J	**70%**	6.60 J	**72%**	5.15 J	**74%**	3.06 J	**62%**	8.95 J	**66%**
# (Millions) Lines Transferred From Memory												
BVH2	24.7 M	**100%**	32.9 M	**100%**	86.2 M	**100%**	76.4 M	**100%**	36.3 M	**100%**	143.3 M	**100%**
BVH2$^+$	24.7 M	**100%**	32.9 M	**100%**	86.1 M	**100%**	76.0 M	**99%**	36.2 M	**100%**	143.0 M	**100%**
BVH4	20.1 M	**81%**	27.7 M	**84%**	71.8 M	**83%**	66.8 M	**87%**	28.7 M	**79%**	117.7 M	**82%**
BVH4$^+$	20.0 M	**81%**	27.6 M	**84%**	71.5 M	**83%**	66.7 M	**87%**	28.5 M	**79%**	116.8 M	**82%**
BVH8	19.2 M	**78%**	27.5 M	**84%**	72.5 M	**84%**	71.7 M	**94%**	27.7 M	**76%**	114.3 M	**80%**
BVH8$^+$	19.1 M	**77%**	27.5 M	**84%**	72.7 M	**84%**	72.0 M	**94%**	27.7 M	**76%**	113.6 M	**79%**
DST*	15.4 M	**62%**	22.9 M	**70%**	60.2 M	**70%**	54.7 M	**72%**	22.7 M	**63%**	92.1 M	**64%**
DST	15.2 M	**61%**	22.8 M	**69%**	60.1 M	**70%**	54.3 M	**71%**	22.5 M	**62%**	89.6 M	**63%**

$^+$ *BVH with extended parallel ray-box pipelines*
* *Dual-split tree with software decoding*

A breakdown of energy consumption is provided in Table 6. Notice that the overall energy consumption comes mainly from DRAM and caches. Both versions of dual-split trees reduce all three types of energy as compared to different kinds of BVH.

Similar to dual-split trees, the speedup of BVH4 and BVH8 grows with the scene complexity. By skipping levels, wider BVHs reduce the acceleration size and memory traffic during traversal. On the other hand, wider BVHs generally have more computational cost, due to more ray-box intersection tests and additional operations for child node ordering. As shown in Table 5, when the scene has low complexity, the memory savings of BVH4 and BVH8 might not outweigh the additional computational costs. Using parallelized ray-box pipelines helps reduce the computational cost of BVH4 and BVH8, which can lead to significant performance improvement (up to 1.23× speedup for BVH8 and up to 1.20× speedup for BVH4 in path tracing task). But when the scene becomes large and complex, the benefit of parallelized ray-box intersections diminishes. For example, in the

Table 6. *Path tracing energy consumption breakdown.*

	Crytek Sponza			Hairball		
	DRAM	Caches	Compute	DRAM	Caches	Compute
BVH2	1.55 J	3.70 J	0.16 J	4.19 J	2.68 J	0.11 J
BVH2$^+$	1.49 J	3.58 J	0.16 J	4.17 J	2.62 J	0.11 J
BVH4	1.49 J	3.69 J	0.16 J	3.84 J	2.68 J	0.11 J
BVH4$^+$	1.35 J	3.48 J	0.16 J	3.66 J	2.58 J	0.11 J
BVH8	1.58 J	4.23 J	0.17 J	3.99 J	3.23 J	0.13 J
BVH8$^+$	1.37 J	3.95 J	0.17 J	3.51 J	3.07 J	0.13 J
DST*	1.32 J	3.03 J	0.09 J	3.18 J	2.53 J	0.08 J
DST	1.06 J	2.22 J	0.09 J	3.05 J	2.01 J	0.08 J

$^+$ *BVH with extended parallel ray-box pipelines*
* *Dual-split tree with software decoding*

largest scene, San Miguel, the path tracing results only see 1.02× and 1.03× speedup for BVH4$^+$ and BVH8$^+$ over their counterparts with non-parallelized ray-box intersections.

Hardware-accelerated dual-split trees, on the other hand, consistently provide faster render times with significantly less compute units per TM, such as 6×, 12×, and 24× fewer floating point adders and multipliers, as compared to BVH2$^+$, BVH4$^+$, and BVH8$^+$, respectively.

4.5 Parameter Exploration

To see how the performance of different acceleration structures scales with the number of multiprocessors, we experiment with varying the number of TMs connected to the L2 cache. Table 7 shows render times of different acceleration structures in the 32, 64 (default), and 128 TM settings. The reported speedup values of the 64 and 128 TM settings are measured relative to the 32 TM setting. Figure 9 shows the frames per second (FPS) for the same tests.

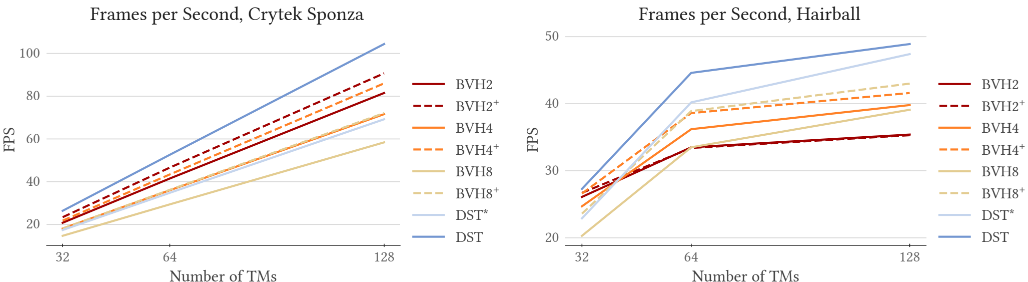


Fig. 9. *Line charts showing frames per second of different acceleration structures in path tracing as the function of number of TMs in Crytek Sponza and Hairball.*

In the Crytek Sponza scene, the smallest scene in our tests, all acceleration structures achieve a similar speedup with increasing TM count and the FPS of all acceleration structures increases linearly with the number of TMs. For the hairball scene however, the performance of different acceleration structures plateaus at different speeds. Notably, for the baseline BVH2 the FPS marginally increases with the number of TMs, which is due to intense resource conflicts during memory load instructions. The fact that the frame rate of the dual split trees continues to grow with the number of TMs indicates that dual split trees are less memory and more compute bound in the architecture being tested.

Table 7. *Relationship between frame time (milliseconds) and number of TMs in path tracing.*

	Crytek Sponza					Hairball				
	32 TM	**64 TM**		**128 TM**		**32 TM**	**64 TM**		**128 TM**	
	Time	Time	speedup	Time	speedup	Time	Time	speedup	Time	speedup
BVH2	48.0 ms	24.1 ms	1.99×	12.3 ms	3.92×	38.4 ms	29.8 ms	1.28×	28.2 ms	1.36×
BVH2+	42.7 ms	21.4 ms	1.99×	11.0 ms	3.87×	37.5 ms	29.9 ms	1.25×	28.3 ms	1.32×
BVH4	55.4 ms	27.8 ms	1.99×	14.0 ms	3.97×	40.5 ms	27.6 ms	1.47×	25.1 ms	1.61×
BVH4+	46.0 ms	23.1 ms	1.99×	11.6 ms	3.95×	37.6 ms	25.9 ms	1.45×	24.0 ms	1.57×
BVH8	67.9 ms	34.1 ms	1.99×	17.1 ms	3.97×	49.3 ms	29.8 ms	1.65×	25.6 ms	1.93×
BVH8+	55.1 ms	27.6 ms	2.00×	13.9 ms	3.97×	42.4 ms	25.7 ms	1.65×	23.2 ms	1.82×
DST*	57.4 ms	28.8 ms	2.00×	14.4 ms	3.97×	43.8 ms	24.9 ms	1.76×	21.1 ms	2.07×
DST	37.9 ms	19.0 ms	1.99×	9.6 ms	3.96×	36.6 ms	22.4 ms	1.63×	20.4 ms	1.79×

+ *BVH with extended parallel ray-box pipelines*
* *Dual-split tree with software decoding*

When the scene is dominated by memory cost, as is in the case of Hairball, a considerable portion of threads wait for memory reads at any given time, limiting the number of threads that can successfully issue instructions per cycle (i.e. avoid stalls). As a result, acceleration structures that are less memory bound (such as dual-split trees and BVH8) can use the computational resources more effectively.

5 DISCUSSIONS AND FUTURE WORK

Dual-split trees provide an attractive alternative to wider BVHs for reducing the storage of binary BVHs as the structure uses less memory and compute as demonstrated in our results. Yet, dual-split trees might benefit less from lossy compression as wider BVHs [Ylitie et al. 2017], due to the fact that plane information has a relatively small portion in the node structure. A possible solution is to collapse the tree levels as suggested in Lin et al. [2019a] to reduce the number of nodes. This is at the expense of potentially reading in more planes than necessary as in the case of high arity BVHs. Nevertheless, unquantized dual-split trees show much higher performance than unquantized BVHs when implemented with hardware acceleration. Thus, it is reasonable to speculate that quantized dual-split trees with collapsed levels would be at least faster than quantized and collapsed wide BVHs on the same hardware. Future work can find out whether quantization of dual-split trees with collapsed levels provides the same amount of speedup for dual split trees as compared with the case for BVHs.

We have observed that dual-split trees are 1.27 - 1.46 × faster than binary BVHs using TRaX as a generic parallel architecture. This is more significant than the findings on software-only CPU implementation [Lin et al. 2019a], where dual-split trees have on average 1.17× speedup over binary BVHs.

While the proposed specialized hardware pipeline substantially reduces the cost of using a dual-split tree, another important factor is that the memory behavior of the tested ray tracing architecture is different from common CPUs. The memory performance is less affected by making more jumps and fetching more but smaller nodes. Additionally, the highly parallel nature of the tested hardware allows for effective sharing of scene data thus amortizing the cost of memory accesses.

While GPUs also have thousands of threads, the SIMD organization of warps requires the memory accesses to be highly aligned such that fewer but larger memory transactions are preferred. With the recent addition of hardware acceleration for ray tracing in the NVIDIA Turing RTX architecture, the ability of dual-split trees to become mainstream entirely depends on whether the memory

behavior can map well onto that hardware. This is something that cannot be explored without detailed simulation models of the RTX hardware.

6 CONCLUSION

We have introduced hardware-accelerated dual-split trees and demonstrated their advantages using detailed simulations on a general parallel hardware ray tracing architecture. Our simulation results show that dual-split trees with specialized hardware intersection units can achieve significantly faster performance than BVHs with different arities using hardware-accelerated ray-box intersection tests in a variety of tested scenes. We have also compared with a software implementation of the intersection logic and have shown that hardware acceleration is extremely effective in attaining higher performance. In addition to performance improvements, a hardware-accelerated dual-split tree also substantially reduces the energy consumption compared to the software implementation and various kinds of BVHs. Both improving performance and lowering energy cost is a rare combination of results.

REFERENCES

Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 14.

Carsten Benthin, Ingo Wald, Sven Woop, and Attila T. Áfra. 2018. Compressed-leaf Bounding Volume Hierarchies. In *High-Performance Graphics (HPG '18)*. 1–4.

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. *USIMM: the utah simulated memory module.* Technical Report. University of Utah.

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 1225–1233.

Michael J Doyle, Colin Fowler, and Michael Manzke. 2013. A hardware unit for fast SAH-optimised BVH construction. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–10.

Michael J Doyle, Ciaran Tuohy, and Michael Manzke. 2017. Evaluation of a BVH construction accelerator architecture for high-quality visualization. *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)* 4, 1 (2017), 83–94.

Bartosz Fabianowski and John Dingliana. 2009. Compact BVH storage for ray tracing and photon mapping. In *Proceedings of Eurographics Ireland Workshop*. 1–8.

Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward A Multicore Architecture for Real-time Ray-tracing. In *41st IEEE/ACM International Symposium on Microarchitecture*.

Christiaan P Gribble and Karthik Ramani. 2008. Coherent ray tracing via stream filtering. In *2008 IEEE Symposium on Interactive Ray Tracing (IRT '08)*. 59–66.

Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. 2006. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing (IRT '06)*. IEEE, 71–80.

Sean Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics (HPG '14)*.

John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. 2009. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *International Symposium on Computer Architecture (ISCA '09)*.

Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2010a. Reconfigurable mobile stream processor for ray tracing. In *IEEE Custom Integrated Circuits Conference 2010 (CICC '10)*.

Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2012. MRTP: Mobile Ray Tracing Processor With Reconfigurable Stream Multi-Processors for High Datapath Utilization. *IEEE Journal of Solid-State Circuits (JSSC)* 47, 2 (2012), 518–535.

Tae-Joon Kim, Bochang Moon, Duksu Kim, and Sung-Eui Yoon. 2010b. RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* 16 2 (2010).

Daniel Kopta. 2016. *Ray tracing from a data movement perspective.* Ph.D. Dissertation. The University of Utah.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2013. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of High-Performance Graphics (HPG '13)*. 121–128.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2015. Memory Considerations for Low Energy Ray Tracing. *Computer Graphics Forum* 34, 1 (2015), 47–59.

Daniel Kopta, Josef Spjut, Erik Brunvand, and Alan Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD '10)*.

Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. 109–119.

Alexander Lier, Magdalena Martinek, Marc Stamminger, and Kai Selgrad. 2018. A High-Resolution Compression Scheme for Ray Tracing Subdivision Surfaces with Displacement. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2 (Aug. 2018), 1–17.

Gábor Liktor and Karthik Vaidyanathan. 2016. Bandwidth-efficient BVH Layout for Incremental Hardware Traversal. In *Proceedings of High Performance Graphics (HPG '16)*. 51–61.

Daqi Lin, Konstantin Shkurko, Ian Mallett, and Cem Yuksel. 2019a. Dual-split trees. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '19)*. 1–9.

Daqi Lin, Konstantin Shkurko, Ian Mallett, and Cem Yuksel. 2019b. Dual-split trees–supplemental materials. (2019).

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

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

Karthik Ramani and Christiaan Gribble. 2009. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*.

Jörg Schmittler, Ingo Wald, and Philipp Slusallek. 2002. SaarCOR – A Hardware Architecture for Realtime Ray-Tracing. In *EUROGRAPHICS Workshop on Graphics Hardware*.

Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. 2004. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Graphics Hardware (GH '04)*. 95–106.

Kai Selgrad, Alexander Lier, Magdalena Martinek, Christoph Buchenau, Michael Guthe, Franziska Kranz, Henry Schäfer, and Marc Stamminger. 2016. A Compressed Representation for Ray Tracing Parametric Surfaces. *ACM Transactions on Graphics (TOG)* 36, 1 (Nov. 2016).

Konstantin Shkurko, Tim Grant, Erik Brunvand, Daniel Kopta, Josef Spjut, Elena Vasiou, Ian Mallett, and Cem Yuksel. 2018. SimTRaX: Simulation Infrastructure for Exploring Thousands of Cores. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI (GLSVLSI)*. 503–506.

Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual Streaming for Hardware-accelerated Ray Tracing. In *Proceedings of High Performance Graphics (HPG '17)*.

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 28, 12 (2009).

Josef Spjut, Daniel Kopta, Solomon Boulos, Spencer Kellis, and Erik Brunvand. 2008. TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *IEEE Symposium on Application Specific Processors (SASP)*.

Elena Vasiou, Konstantin Shkurko, Erik Brunvand, and Cem Yuksel. 2019. Mach-RT: A Many Chip Architecture for High-Performance Ray Tracing. In *High-Performance Graphics (HPG '19)*. ACM, New York, NY, USA.

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2017. MergeTree: A fast hardware HLBVH constructor for animated ray tracing. *ACM Transactions on Graphics (TOG)* 36, 5 (2017), 1–14.

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Aleksi Tervo, and Jarmo Takala. 2018. PLOCTree: A Fast, High-Quality Hardware BVH Builder. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2 (2018), 1–19.

Carsten Wächter. 2008. *Quasi-Monte Carlo light transport simulation by efficient ray tracing.* Ph.D. Dissertation. Universität Ulm.

Carsten Wächter and Alexander Keller. 2006. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques* 2006 (2006), 139–149.

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs. In *Symposium on Interactive Ray Tracing (IRT '08)*. 49–57.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.

Sven Woop, Erik Brunvand, and Philipp Slusallak. 2006a. Estimating Performance of a Ray Tracing ASIC Design. In *Interactive Ray Tracing (IRT '06)*.

Sven Woop, Gerd Marmitt, and Philipp Slusallek. 2006b. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware (GH '06)*. 67–77.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics (TOG)* 24, 3 (July 2005).

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics (HPG '17)*. 1–13.

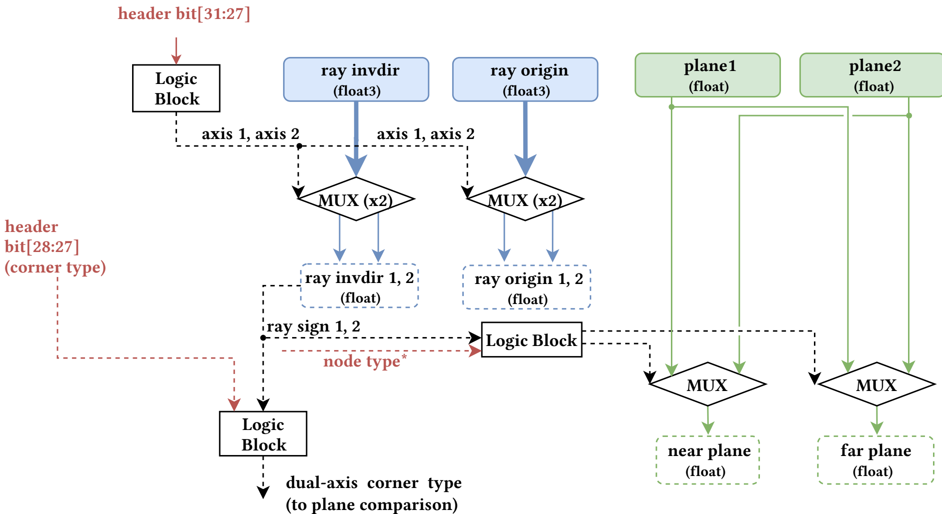


Fig. 10. *The detailed view of the ray component and plane selection logic. Header data and derived data from header (marked with asterisks) are colored using dark red. Round boxes with dashed outline represent temporary data which are consumed by the next stage of pipeline. Relative areas of the units in the diagram do not represent the actual relative areas.*

A RAY COMPONENT AND PLANE SELECTION LOGIC

At the center of the dual-split pipeline there are two FPADD units and two FPMUL units, which compute the intersection distances between the ray and the two planes. Before that, a *ray component and plane selection logic (Figure 10)* is responsible for selecting 1 or 2 ray components from a 3D vector corresponding to the splitting or carving axes, as well as assigning the planes as "near" or "far" according to the 1D (splitting node or single-axis carving node) or 2D (dual-axis carving node) ray direction. To accommodate both 1D and 2D cases, two multiplexers select two components from the 3D ray invdir input, same for ray origin (Figure 10). Notice that the two components are from the same axis in case of the splitting node or the single-axis carving node (1D case), where the resulting ray sign is used to swap the two input planes. In the case of a splitting node, the ray direction sign is also used by the later offset computation logic to assign the offset of the left and right child (in the builder's view) to the near and far child (in the ray view). For the dual-axis carving node (2D case), the input plane order does not change and the ray signs are used to determine the "corner type" (Figure 2) stored in bits[28:27] with respect to the ray direction. This "corner type" is used as the input to the later plane comparison logic which selects the correct planes to compare. Throughout this process the node header and its derived flags (color-coded as dark red in Figure 10, which is also the case in Figure 11 and Figure 12) are used as inputs to simple combinational logic blocks that produce bits to select the multiplexer inputs. An example derived flag is "node type" which uses 2 bits to represent one of the splitting node, single-axis carving node, dual-axis carving node, or leaf node. The purpose of these derived flags is to simplify the combinational logic by using the previously computed results.

B PLANE COMPARISON AND RETURN VALUE LOGIC

Figure 11 shows the plane comparison and return value logic, which is responsible for generating the new (t_{min}, t_{max}) range and the 2-bit return value, derived from the plane comparison results and the node type. As shown in Figure 6, the floating point units produce t_1 and t_2, the ray-plane

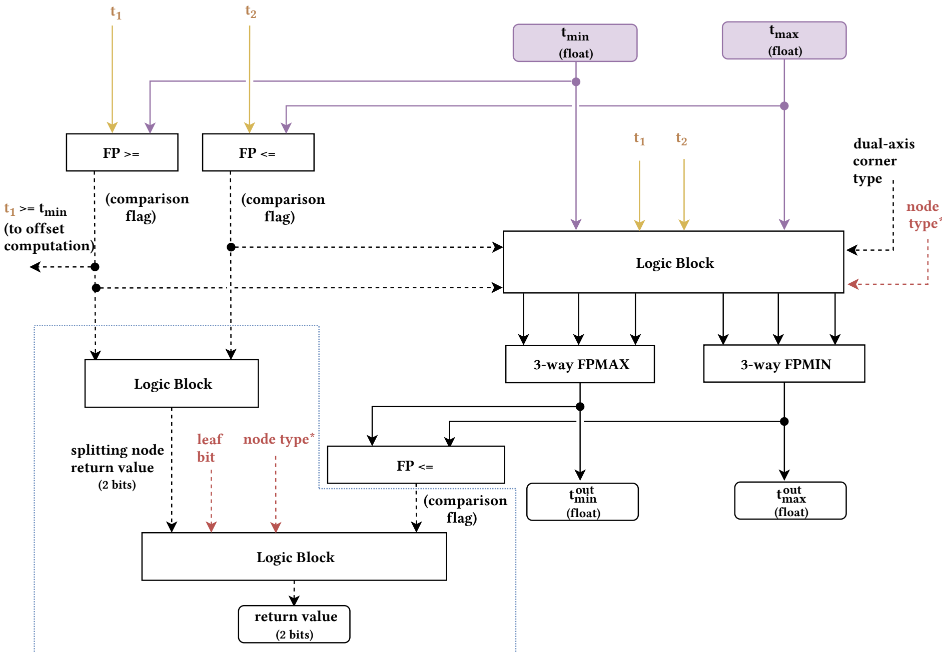


Fig. 11. *The detailed view of the plane comparison and return value logic. The blue dashed line circles the return value logic. Relative areas of the units in the diagram do not represent the actual relative areas.*

intersection distances. These are used as inputs to the plane comparison logic. Whether the ray passes through the empty space and how many children the ray intersects depends on the node type and the comparison between the current ray range (t_{min}, t_{max}) and the intersection distances. The four different intersection cases of a splitting node are illustrated in Figure 3. Notice that the case is entirely determined by the two boolean values, $t_{min} \leq t_1$ and $t_{max} \geq t_2$. The boolean values are passed through a simple logic block that produces 2-bit splitting node return value (number of children intersected), as an intermediate output to produce the final return value. Note that $t_1 \geq t_{min}$ is used as a signal to guide the offset computation. The near child is intersected and used as the next node to traverse only when $t_1 \geq t_{min}$, and the far child is assigned as the node to push to stack; otherwise, the far child is used as the next node to traverse. For the splitting node, the t^{out}_{min} and t^{out}_{max} outputs are used to trim the ray range in the intersected children. They serve as the new ray range when only one child is intersected; when both children are intersected, t^{out}_{min} and t^{out}_{max} become the t_{min} and t_{max} of the far and the near child, respectively, since they both share the other end of ray range with the parent. The determination of t^{out}_{min} and t^{out}_{max} depends on the intersection case and comparing t_1, t_2 with the original t_{min} and t_{max}. This is done in a combination of the floating point comparison units and the floating point min/max units.

For carving nodes, t^{out}_{min} and t^{out}_{max} represent the trimmed range. Apparently, when $t^{out}_{min} > t^{out}_{max}$ the ray is culled. This condition is determined by a floating point comparison unit (FP$\leq$ in the figure) and passed as an input to the final logic block to determine the return value. In the carving node case, apart from 0 (no intersection) and 1 (has intersection), there is also a special case of 3, which corresponds to an intersected carving node acting as a leaf node. As shown in Figure 4, when the lowest bit in the 6-bit header is 1 in the carving node case, it indicates that the carving node directly stores the triangle offset instead of a pointer to a standalone leaf node, which is a technique to avoid

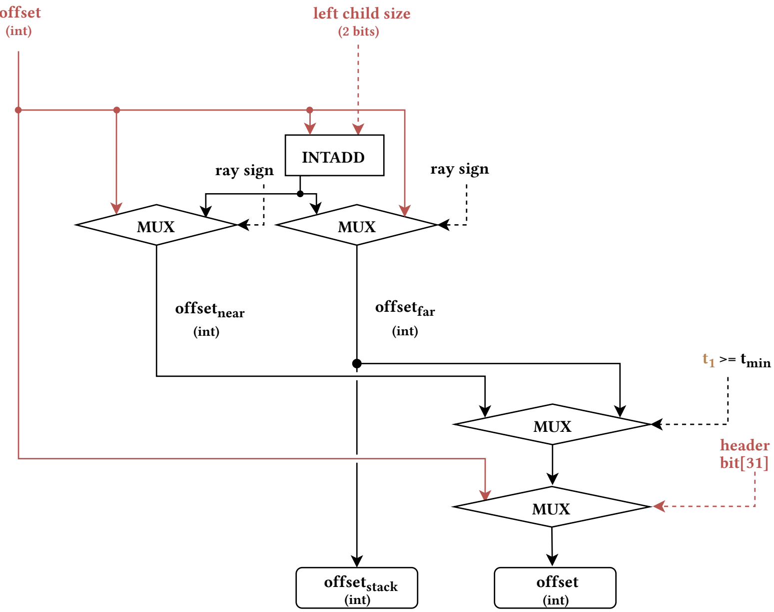


Fig. 12. *The detailed view of the offset computation logic. Relative areas of the units in the diagram do not represent the actual relative areas.*

wasting storage, by taking advantage that a carving node only has one child. Correspondingly, the leaf bit is extracted from the header as an input to the last logic block. The derived node flag is also connected to the logic block to choose the intermediate return type according to the node type.

Notice that we use 3-way FPMAX and FPMIN units to find t_{min}^{out} and t_{max}^{out} because unlike splitting nodes or single-axis carving nodes that compute each of t_{min}^{out} and t_{max}^{out} from two values, dual-axis carving nodes have two cases shown in Figure 2a and Figure 2d that require finding the minimum or maximum of three values. For example, in Figure 2a, t_{max}^{out} is chosen as the minimum of the three values t_1, t_2, and t_{max}. The case of t_{max} (not shown in Figure 2a) corresponds to the case of the original t_{max} being inside the non-empty region defined by the carving node. In this case, finding the maximum only from t_1 and t_2 will incorrectly enlarge the ray range. The case for Figure 2d is similar. Since the FPMAX and FPMIN units are shared by all node types, a logic block is responsible for taking all possible planes (t_1, t_2, t_{min}, and t_{max}) and information including the node type and the dual carving node corner type to produce the correct 3-way inputs.

C OFFSET COMPUTATION LOGIC

Figure 12 shows how the offsets for the near and far child nodes of a splitting node can be computed using a simple logic block, based on the offset of the left child and the size of the left child encoded in bits [25:0] and [28:27] of the header, respectively. An integer addition unit calculates the right child offset by adding the left child size to the left child offset. The left and right child offsets are assigned to the near and far child by the ray sign through multiplexers. After that, a result from the plane comparison logic can assign the far child offset as the output offset for the next node to traverse when only the far child is intersected. Finally, the output offset is set to the original offset in the header to address the leaf triangles, when the node is not a splitting node.