Timo Viitanen

Hardware Accelerators for Animated Ray Tracing

Julkaisu 1551 • Publication 1551

Tampere 2018

Tampereen teknillinen yliopisto. Julkaisu 1551
Tampere University of Technology. Publication 1551

Timo Viitanen

Hardware Accelerators for Animated Ray Tracing

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 25th of May 2018, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2018

Doctoral candidate: Timo Viitanen
Laboratory of Pervasive Computing
Faculty of Computing and Electrical Engineering
Tampere University of Technology
Finland

Supervisor: Jarmo Takala, Prof.
Laboratory of Pervasive Computing
Faculty of Computing and Electrical Engineering
Tampere University of Technology
Finland

Instructor: Pekka Jääskeläinen, D.Sc. (Tech.)
Laboratory of Pervasive Computing
Faculty of Computing and Electrical Engineering
Tampere University of Technology
Finland

Pre-examiners: Gwo-Giun Lee, Prof.
Department of Electrical Engineering
National Cheng Kung University
Taiwan

Timo Aila, D.Sc. (Tech.)
NVIDIA
Finland

Opponent: Samuli Laine, D.Sc. (Tech.)
NVIDIA
Finland

ISBN 978-952-15-4151-3 (printed)
ISBN 978-952-15-4160-5 (PDF)
ISSN 1459-2045

Abstract

Future graphics processors are likely to incorporate hardware accelerators for real-time ray tracing, in order to render increasingly complex lighting effects in interactive applications. However, ray tracing poses difficulties when drawing scenes with dynamic content, such as animated characters and objects. In dynamic scenes, the spatial datastructures used to accelerate ray tracing are invalidated on each animation frame, and need to be rapidly updated. Tree update is a complex subtask in its own right, and becomes highly expensive in complex scenes. Both ray tracing and tree update are highly memory-intensive tasks, and rendering systems are increasingly bandwidth-limited, so research on accelerator hardware has focused on architectural techniques to optimize away off-chip memory traffic. Dynamic scene support is further complicated by the recent introduction of compressed trees, which use low-precision numbers for storage and computation. Such compression reduces both the arithmetic and memory bandwidth cost of ray tracing, but adds to the complexity of tree update.

This thesis proposes methods to cope with dynamic scenes in hardware-accelerated ray tracing, with focus on reducing traffic to external memory. Firstly, a hardware architecture is designed for linear bounding volume hierarchy construction, an algorithm which is a basic building block in most state-of-the-art software tree builders. The algorithm is rearranged into a streaming form which reduces traffic to one-third of software implementations of the same algorithm. Secondly, an algorithm is proposed for compressing bounding volume hierarchies in a streaming manner as they are output from a hardware builder, instead of performing compression as a postprocessing pass. As a result, with the proposed method, compression reduces the overall cost of tree update rather than increasing it. The last main contribution of this thesis is an evaluation of shallow bounding volume hierarchies, common in software ray tracing, for use in hardware pipelines. These are found to be more energy-efficient than binary hierarchies. The results in this thesis both confirm that dynamic scene support may become a bottleneck in real time ray tracing, and add to the state of the art on tree update in terms of energy-efficiency, as well as the complexity of scenes that can be handled in real time on resource-constrained platforms.

Preface

The work described in this Thesis took place over 2013-2018 in the Department of Pervasive Computing, Tampere University of Technology, Finland, and in part during my research visit to the DSPCAD group, University of Maryland, USA.

I am thankful to Prof. *Jarmo Takala* for the opportunity to pursue this research, and all his guidance and support. The guidance of Dr. *Pekka Jääskeläinen* was also invaluable: he encouraged me in taking up this intriguing and complex research topic, and seeing the work through despite a variety of distractions. Moreover, I am grateful for my thesis pre-examiners Dr. *Timo Aila* and Prof. *Gwo-Giun Lee* for their valuable comments. Prof. *Shuvra Bhattacharyya* has my special thanks for arranging an opportunity to visit his research group in Maryland.

It has been a joy to work with the many friends and colleagues in Jarmo's research group. I would like to thank, in particular, my closest collaborators in this work, *Matias Koskela*, M.Sc., *Heikki Kultala*, M.Sc. and *Kalle Immonen*, M.Sc., for their helpful assistance, insightful comments, and many fruitful discussions. Many others in the group also helped in this work and contributed to a lively, creative atmosphere. In addition, I would like to express special thanks to Prof. Bhattacharyya's research group at Maryland and in particular *Lin Li*, B.Sc., and *Kyunghun Lee*, B.Sc.

I gratefully acknowledge the financial support that allowed me to work on this interesting topic. In the past four years, I have received support from the *TUT Graduate School*, the *EU commission* via the ARTEMIS joint undertaking ALMARVI, the *National Technology Agency of Finland (TEKES)*, and the *Nokia Foundation.*

Finally, I am thankful to my parents *Ulla* and *Oiva*, and siblings *Tuula* and *Tapio* for their constant support.

Contents

Abstract **iii**

Preface **v**

Acronyms **ix**

Nomenclature **xi**

List of Publications **xiii**

1 Introduction **1**
1.1 Scope and Objectives of Research 4
1.2 Main Contributions 5
1.3 Author's Contribution 5
1.4 Thesis Outline 6

2 Streaming Linear BVH Construction **7**
2.1 BVH Construction Methods 8
2.1.1 Linear BVH 10
2.1.2 Refinement Based Construction 12
2.1.3 Spatial Splits 13
2.1.4 Other Build Algorithms 14
2.1.5 Refitting 15
2.1.6 Summary 15
2.2 Hardware Accelerated Construction 15
2.2.1 Binned SAH Acceleration 16
2.2.2 Refit Acceleration 16
2.2.3 k-Dimensional Tree Accelerators 17
2.2.4 Imagination Technologies Builder 17
2.3 Sorting Hardware 18
2.4 Thesis Contribution 19

3 Rebuilding and Refitting Compressed BVHs **23**
3.1 BVH Compression Methods 24
3.1.1 Coordinate Compression 24
3.1.2 Pointer Compression 26
3.1.3 Primitive Compression 27
3.1.4 Entropy Coding 29
3.1.5 Comparison 29

3.2 Incremental Encoding 30
3.3 Thesis Contribution 32

4 Hardware-Accelerated Shallow BVHs **33**
4.1 Traversal Architectures 34
4.1.1 Programmable Platforms 34
4.1.2 Fixed-Function Accelerators 35
4.1.3 Memory Access Schemes 35
4.2 Multi-Bounding Volume Hierarchies 37
4.2.1 Construction 39
4.2.2 Traversal 39
4.3 Thesis Contribution 40

5 Conclusion **43**
5.1 Main Results 44
5.2 Open Research Issues 45

Bibliography **47**

Publications **59**

Acronyms

AABB Axis Aligned Bounding Box

AAC Approximate Agglomerative Construction

AR Augmented Reality

ASIC Application Specific Integrated Circuit

ATRBVH Agglomerative Treelet Restructuring Bounding Volume Hierarchy

AVX Advanced Vector Extensions

B-KD tree Bounded k-dimensional tree

BIH Bounding Interval Hierarchy

BRAM Block Random Access Memory

BSAHA Binned SAH Accelerator

BSP Binary Space Partition

BVH Bounding Volume Hierarchy

CAD Computer Aided Design

CBVH Compressed Bounding Volume Hierarchy

CMBVH Compressed Multi Bounding Volume Hierarchy

CPU Central Processing Unit

DE-tree Dual Extent tree

DRAM Dynamic Random Access Memory

FIFO First In First Out

FoV Field of View

FPGA Field Programmable Gate Array

FPU Floating Point Unit

GTU Geometry and Tree Update unit

HMQ Hierarchical Mesh Quantization

IOSP Implicit Object Space Partition

GPGPU General-Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HCCMesh Hierarchical Culling oriented Compact Mesh

HLBVH Hierarchical Linear Bounding Volume Hierarchy

***k*-d tree** *k*-dimensional tree

LBVH Linear Bounding Volume Hierarchy

LFD Light Field Display

LoD Level of Detail

MBVH Multi Bounding Volume Hierarchy

MIC Many Integrated Cores

MIMD Multiple Instruction Multiple Data

MVH Minimal Bounding Volume Hierarchy

NURBS Non Uniform Rational Basis Spline

QBVH Quad Bounding Volume Hierarchy

RACBVH Random Accessible Compressed Bounding Volume Hierarchy

RAU Ray Accumulation Unit

RBVH Rasterized Bounding Volume Hierarchy

RTL Register Transfer Level

SAH Surface Area Heuristic

SBVH Split Bounding Volume Hierarchy

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

sk-d tree Spatial K-Dimensional tree

SMT Simultaneous Multi-Threading

SPBVH Shared Plane Bounding Volume Hierarchy

SRAM Static Random Access Memory

SSE Streaming SIMD Extensions

TBU Tree Build Unit

TRBVH Treelet Restructuring Bounding Volume Hierarchy

VR Virtual Reality

Nomenclature

Ray traversal (Chapter 2):

x	Ray-scene intersection point
o	Ray origin
d	Ray direction
t	Parametric distance to intersection, $x = o + dt$

Surface Area Heuristic (Section 2.2.1):

C	Surface Area Heuristic cost of a BVH tree
C_{node}	Heuristic cost of traversing an inner node
C_{leaf}	Heuristic cost of traversing a leaf
C_{tri}	Heuristic cost of a primitive intersection test
$A(N_i)$	Surface area of inner node i
$A(L_i)$	Surface area of leaf node i
$A(R)$	Surface area of tree bounding box
n_{nodes}	Number of nodes in tree
n_{leafs}	Number of leafs in tree

Binned SAH construction (Section 2.2.1):

S	Number of candidate splits per axis in binned SAH construction

External mergesort (Section 2.3):

N	Sort input size
M	Size of fast local memory
B	Minimum block size for transfers between local and external memory
k	Multi-merge width

Heap layout (Section 3.1.2):

i	Element index
K	Tree branching factor

Triangle strips (Section 3.1.3):

v_i	ith vertex of a triangle strip
n	Number of vertices in a strip.

Incremental encoding (Section 3.2):

b	Number of bits used to store an AABB coordinate
r_x, r_y, r_z	Lower bound offsets relative to parent AABB
s_x, s_y, s_z	Upper bound offsets relative to parent AABB

List of Publications

This Thesis consists of an introductory part and four original publications. In the text, these publications are referred to as [P1] through [P4].

[P1] Viitanen T., Koskela M., Jääskeläinen P., Kultala H., Takala J.: MergeTree: a HLBVH constructor for mobile systems. In *SIGGRAPH Asia Technical Briefs* (2015), p. 12.

[P2] Viitanen T., Koskela M., Jääskeläinen P., Kultala H., Takala J.: MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing. *ACM Transactions on Graphics 36*, 5 (2017), 169.

[P3] Viitanen T., Koskela M., Jääskeläinen P., Immonen K., Takala J.: Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies. *Computer Graphics Forum 36*, 4 (2017), 167−178.

[P4] Viitanen T., Koskela M., Jääskeläinen P., Takala J.: Multi Bounding Volume Hierarchies for Ray Tracing Pipelines. In *SIGGRAPH Asia Technical Briefs* (2016), p. 8.

1 Introduction

Ray tracing is the future and ever will be.

–- David Kirk

Computer-generated images are ubiquitous in modern society and form a main component of human-computer interaction. Moreover, computer graphics are applied to entertainment, e.g., video games and motion pictures, at a grand scale. As a result, great effort has gone to optimizing rendering performance, allowing increasingly smooth and high-quality visuals in real-time applications, and offline rendering almost indistinguishable from photographs. Recently, rendering systems are further challenged by emerging consumer-grade Virtual Reality (VR) and Augmented Reality (AR), which require images to be drawn at very high resolutions and frame rates for an immersive experience. Moreover, a large amount of rendering is done in mobile devices, which have tight energy constraints. The combination of mobile AR in particular is predicted to open applications of high societal impact in, e.g., industry, medicine, entertainment and education [vKP10]. Since rendering for mobile AR combines the tight energy budgets of mobile platforms with the performance requirements of high-quality VR, it may require novel architectural approaches.

Most real-time and interactive rendering is performed by means of *rasterization* and the z-buffer algorithm, accelerated by Graphics Processing Units (GPUs). Given a 3D scene made up of geometric primitives − usually triangles − and a camera point, the rasterization approach performs geometric transforms to project each primitive to a screen coordinate frame, and then *shades* all pixels of the resulting 2D primitive to decide their color. The z-buffer algorithm [Cat74] is used to handle occlusions between primitives. In the first GPUs, shading was performed with a fixed-function pipeline with a limited selection of operating modes, but is now fully programmable. As a result, the GPUs of today are massively parallel general-purpose computation engines used to accelerate many applications beyond graphics.

A long-standing goal in computer graphics research is to replace or augment rasterization in real-time rendering with *ray tracing*, an alternative approach where it conceptually easier to model many lighting effects realistically. A basic operation in ray tracing is *ray traversal*, i.e., determining the closest hit point between a ray and a 3D scene. At its simplest, this operation is used to traverse *primary rays* for each pixel starting from the camera origin. Various visual effects can then be modeled by traversing *secondary rays* that start with the intersection point. For example, reflections are handled by casting a *reflection ray* which is a mirror image of the primary ray, or shadows are drawn by casting *shadow rays* at light sources to test whether they are occluded.

Figure 1.1: Example of a path traced image with shadows, glossy materials and indirect illumination. Scene by Jay-Artist, CC BY 3.0.

Other effects can be rendered by casting a set of rays drawn from a random distribution: doing this with, e.g., primary, reflection and shadow rays gives focal blur, glossy reflections and soft shadows, respectively [CPC84]. Photorealistic images can be generated with *path tracing* [Kaj86] which emulates the physical behavior of light by integrating over many secondary rays which act as pseudo-photons, rebounding in the scene at random until they reach a light source. An image rendered this way is at first noisy, but eventually converges to a physically correct distribution. Path tracing is the gold standard of photorealistic rendering and used to render, e.g., the animated motion pictures of Disney and Pixar [KFF*15]. Figure 1.1 shows an example of a path traced scene.

A ray tracing system uses ray traversal and two other primitive operations, *tree build* and *shading*, as building blocks for higher-level rendering algorithms. Briefly, tree builds are used to adapt the acceleration data structure used in ray traversal to changes in the scene between animation frames, while shader programs model material appearance. Over the past decade, the performance of all three operations has improved sharply by taking advantage of the growing capabilities of GPU and multicore Central Processing Unit (CPU) hardware. Moreover, an extensive literature has developed optimizing ray tracing at the algorithm level, e.g., by carefully selecting the traversed rays with adaptive sampling, and filtering away the noise caused by low sample counts [ZJL*15]. As a result, current ray tracers are capable of rendering some effects in real time.

A classic motivation for ray tracing has been the above conceptual simplicity of generating visual effects that require complex workarounds in rasterization-based systems. In addition, some works on ray tracing are motivated by its logarithmic scaling with regards to the number of primitives: with sufficiently large scenes, it may outperform rasterization, which has linear complexity [SKBD12]. However, in practice, the complexity of rasterization can be kept in check with visibility determination algorithms [COCSD03]. Rather than displacing rasterization completely, real-time ray tracing is more likely to be used as part of a hybrid system which combines both techniques [PBMH02]. Several applications have been proposed for video games [FGD*06, Bik07].

The recent push for commercial VR and AR seems to strengthen the case for ray tracing. In VR headsets, a rasterization pipeline needs to first draw a planar image and then resample it to compensate for lens distortion. Moreover, the work done to render virtual

scenes can be sharply reduced by using fewer samples in the peripheral vision, based on head or gaze tracking. In ray tracing it is convenient to build lens distortion into ray generation [PJB13], and to sample the area of accurate vision more densely [WRK*16].

There is recent interest in near-eye Light Field Displays (LFDs) for VR which can control the direction of the emitted light, allowing the construction of lightweight headsets with a wide Field of View (FoV), and side benefits such as the use of focal depth 3D cues [LL13]. LFD images have to be rendered from many views, which is advantageous for ray tracing. Finally, in AR, it is interesting to create virtual objects which blend seamlessly into the surroundings. To this end, the renderer needs estimates of the shape, materials and illumination of the surrounding scene. Several state-of-the-art methods to produce these estimates make heavy use of ray tracing [RTKPS16, KPR*15].

Recently, there has been growing academic and industrial interest in hardware ray tracers, including a product launch by the major mobile GPU vendor Imagination Technologies [Pow15]. Ray tracing architectures have been proposed ranging from fixed-function hardware pipelines [WSS05, NPP*11, LSL*13], to exotic programmable Multiple Instruction Multiple Data (MIMD) processors [SKKB09, KSS*15], to hybrid architectures integrating ray tracing functionality to conventional GPUs [Kee14]. These architectures form a trade-off curve between performance and flexibility. At one end of the curve, fixed-function accelerators can be said to have the most short-term interest, since they are likely to reach a greater performance in a given energy and silicon area budget, and thus become practical sooner than programmable systems. In some cases, fixed-function pipelines are 2-3 orders of magnitude more efficient than a general-purpose processors [HQW*10], though the available benefits are more limited in a memory-intensive, floating-point heavy application such as ray tracing.

Possibly the strongest motivation for hardware ray tracing is that it may render some complex visual effects at a lower energy cost than multi-pass rasterization based methods [KKW*13]. Today's rendering systems are increasingly limited by the thermal design power and memory bandwidth of the GPU. Previously, the graphics community might have relied on transistor scaling to provide steadily improving performance in a given power budget. However, due to the recent breakdown of Dennardian CMOS scaling, the energy efficiency gains from scaling have diminished, and attention in circuit design is turning to hardware specialization to take advantage of increasing transistor counts [Tay12]. Mobile devices operate under far more stringent power constraints and, as a result, the potential efficiency benefits of ray tracing have spurred the development of several mobile ray tracing acccelerators, which have been proposed as enablers of photorealistic mobile AR and VR [KKK12, NKK*14, LSL*13].

Recent work on ray tracing accelerators has provided promising insights into the rendering process which may allow order-of-magnitude reductions in its computational cost: namely, that the cost of hardware ray tracing is dominated by memory accesses, which can be reduced to a small fraction by rearranging the access pattern [AK10, KSS*15] and compressing the acceleration data structure [Kee14]. Hardware ray tracers are able to reach very high simulated ray traversal throughputs, e.g., giving performances close to a desktop General-Purpose Graphics Processing Unit (GPGPU) in a mobile envelope [LSL*13], or orders of magnitude higher performances in a desktop envelope [Kee14].

Aside from ray traversal, a main component of the ray tracing problem is *tree update*, i.e., supplying the traversal process with a data structure which indexes the scene geometry and accelerates ray traversal. In photorealistic offline rendering, or when visualizing static scenes, the runtime of tree update is of no consequence, and the main consideration is

to produce trees with high *tree quality* which are cheap to traverse. However, when ray tracing animated scenes in real time, tree update must be done before each animation frame – e.g. at 90Hz for recent VR headsets – and doing this fast enough while retaining a reasonable tree quality is a challenging sub-problem. Most applications of interest have animated geometry, e.g., in computer games it is commonplace to display complex particle effects or crowds of detailed characters moving based on *skeletal animation*. Ray tracing accelerators proposed and demonstrated so far have been mostly limited to static or mostly static scenes, ruling out these applications.

1.1 Scope and Objectives of Research

The general subject of this Thesis is to explore techniques for hardware-accelerated ray tracing, and especially the real-time ray tracing of large-scale animated scenes.

A complete ray tracing rendering system combines the low-level tools of tree update, ray traversal and shading to generate various visual effects. This Thesis is limited to implementations of tree update and traversal, ruling out shading and system-level optimizations. It should be noted that shading can rival traversal and construction in terms of computational cost [LKA13], and there is a large literature of high-level optimizations such as denoising [ZJL*15] to reduce the amount of ray traversal operations needed to generate a high-quality image.

Previous work on ray tracing accelerators can be divided into programmable and fixed-function systems. Since fixed-function accelerators are often significantly more efficient than programmable systems, they are more likely to be of practical interest in the short term, and this Thesis focuses on them.

The focus on animated scenes places emphasis on the sub-problem of tree update, i.e., refitting or reconstructing the acceleration data structure. As the data structure of choice, we focus on Bounding Volume Hierarchy (BVH) and certain variants. BVH is presently popular since other structures are impractical to update. Real time BVH update on desktop CPUs and GPGPUs has been the subject of intensive study, and can be regarded as a solved problem. Hence, the work in this Thesis focuses on two open cases.

Firstly, fast tree updates consume a large amount of memory bandwidth, which is then unavailable for rendering and shading. Bandwidth is especially limited in mobile systems, but savings would also be useful on the desktop. Secondly, *compressed trees* are efficient to traverse, but updating them remains an open problem. These points lead to the two main research questions which Thesis answers:

- Can high-performance GPGPU tree construction algorithms be efficiently adapted into a bandwidth economical, hardware-accelerated context?
- Is it possible to reach real-time update rates for Compressed Bounding Volume Hierarchies (CBVHs)?

The objective of this research is to develop effective methods for tree construction and compressed tree update in bandwidth-limited systems.

1.2 Main Contributions

This Thesis proposes techniques to support animated scenes in hardware-accelerated ray tracing, with a focus on energy- and memory-constrained systems such as mobile phones. In [P1,P2], a hardware architecture is proposed for the Linear Bounding Volume Hierarchy (LBVH) tree construction algorithm, which forms the basis for the state-of-the-art high performance GPU tree builders. In order to improve memory bandwidth economy, LBVH is expressed as streaming processes, and an external sorting algorithm is used. The architecture is evaluated extensively with Register Transfer Level (RTL) implementation, power analysis and system-level simulations.

Article [P3] introduces algorithmic techniques for streaming bottom-up compression of BVHs, aimed for a possible hardware implementation. The benefit is that tree update processes can directly output compressed trees, in the best case halving memory traffic. As CBVH nodes are encoded relative to the parent [VAMS16, Kee14], this requires estimating the parent context of each node, and backtracking to repair the hierarchy when the estimated context is wrong. Novel techniques of *modulo encoding* and *treelet-based compression* are proposed to minimize backtracking.

The main drawback of [P3] is that state-of-the-art CBVHs based on *plane sharing* [VAMS16] cannot be compressed. In [P4], we make an initial step toward addressing this shortcoming by showing that Multi Bounding Volume Hierarchies (MBVHs) could be used to save memory traffic in the place of plane sharing.

In summary, the main novel contributions in this Thesis are the following:

- a bandwidth-conserving hardware architecture for LBVH tree construction, which trades off tree quality for major runtime, energy and chip area savings compared to the state-of-the-art hardware builder,
- algorithmic techniques for streaming bottom-up update of CBVHs, which reduce the memory traffic cost of CBVH update by up to half, and
- an evaluation of hardware ray tracing with MBVHs.

1.3 Author's Contribution

This Thesis includes four publications [P1-P4]. [P1] proposes a hardware LBVH build unit. The author invented the key techniques of the architecture of hardware heap based external sorting and streaming hierarchy emission, designed the architecture, wrote a cycle-level simulator, and performed the measurements and analysis of results.

[P2] extends [P1] with a complete hardware implementation and system-level modeling results. The Thesis Author implemented the hardware unit, performed ASIC synthesis and power analysis, modified the cycle-level simulator designed in [P4] for use in system-level energy modeling, and performed the additional measurements and analysis.

[P3] proposes algorithms for streaming bottom-up CBVH compression. The Thesis Author designed the algorithms, extended a software ray tracer with the proposed methods, and performed measurements and analysis.

[P4] proposes the use of MBVHs in hardware ray tracing. For this work, the Thesis Author designed a MBVH traversal unit architecture, wrote a software ray tracer and cycle-level hardware simulator, and performed the measurements and analysis of results.

The Thesis Author was the main author and wrote the main body of text for each publication.

1.4 Thesis Outline

This Thesis consists of five previous publications and an introductory part. Chapter 2 briefly introduces some basic concepts in ray tracing. Chapters 2 − 4 expand on the relation of the Thesis main contributions to the state of the art. Chapter 2 focuses on the LBVH accelerator architecture proposed in [P1,P2], Chapter 3 on the streaming CBVH update algorithm proposed in [P3], and Chapter 4 on the application of MBVH to ray traversal accelerators proposed in [P4]. Finally, Chapter 5 discusses the main results, proposes future work and concludes the introductory part. The publications [P1-P4] then follow.

2 Streaming Linear BVH Construction

Ray tracing renders images of 3D scenes specified as lists of geometric primitives, most often triangles. The basic operation in ray tracing, *ray traversal*, can be defined as follows: given a 3D scene, a ray origin o, and ray direction d, find the intersection x between the ray and the scene,

$$x = o + dt, \tag{2.1}$$

which has the shortest *parametric distance* t from the origin. Figure 2.1 shows an example of ray traversal.

A straightforward way to compute ray traversal is to loop over all geometric primitives, compute intersection tests between the ray and each primitive, and select the intersection with lowest t. However, brute-force traversal quickly is infeasible for nontrivial scenes. Fast ray tracing is based on organizing the primitives with an acceleration data structure optimized for ray traversal. There is a large body of research on such data structures, the main classes being k-dimensional trees (k-d trees), grids, and BVHs. A good, though now somewhat dated overview is given by the state of the art survey of Wald et al. [WMG*09].

Recently, research has converged on the BVH as the data structure of choice, as it both gives good rendering performance and is easy to construct and update [WMG*09]. The focus of this Thesis is on BVHs and certain variants optimized for hardware acceleration. In a BVH, each node subdivides primitives into two disjoint sets, whose bounding volumes are stored. If a traced ray does not intersect a bounding volume, all primitives underneath can be discarded, resulting in logarithmic typical-case complexity for ray traversal. The bounding volumes are nearly always Axis Aligned Bounding Boxs (AABBs) in ray tracing, though more complex volumes have been used in collision detection [LMK17].

Ray traversal in a BVH is typically done in depth-first, stack-based manner. Traversal begins by pushing the root node onto a *traversal stack*. Nodes are then repeatedly popped from the stack and visited. When visiting an inner node, the ray is tested against the child AABBs of the node, and intersecting children are pushed to the stack. Visits to leaf nodes cause ray-triangle tests. Important optimizations in traversal are to push children so that they are visited in a closest-first order, and to perform *depth culling*, i.e., skip traversal of nodes that are farther than the closest intersection found so far. Ray traversal performed this way typically requires a number of node visits and intersection tests that is roughly logarithmic with regard to the scene size.

The main alternatives to BVHs, grids and k-d trees, can be characterized as *spatial split* data structures: they divide the scene's 3D space into subvolumes, and each primitive is referenced from all leaf volumes – k-d tree leafs or gridcells – which overlap that

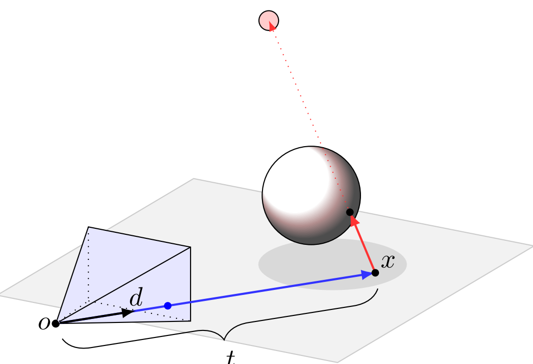


Figure 2.1: Example of ray traversal. o: ray origin, d: ray direction, t: parametric distance, x: intersection point. Red: secondary (shadow) ray.

primitive. BVHs, meanwhile, are *object split* structures where each primitive is referenced once, and the nodes may refer to overlapping volumes. An in-depth comparison of BVH and k-d tree ray traversal is given by Vinkler et al. [VHB14]. In general, BVHs require more floating-point computations, while k-d trees are more memory-intensive.

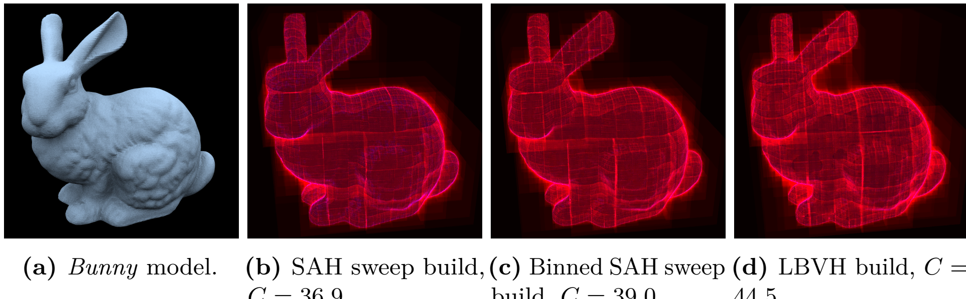
Methods to build or update BVHs have been a subject of extensive study. If a BVH is built once and used for a large amount of rendering work, the main characteristic of interest when choosing the update method is *tree quality*, i.e., how good is the rendering performance given by the resulting tree. In animated, and especially real time rendering, build speed becomes a relevant consideration. The last few years have seen significant advances in GPGPU build algorithms which exhibit both real-time performance and good quality compared to older gold-standard algorithms. A main contribution of this Thesis is a streaming, hardware-oriented build algorithm, which is a step toward bringing these advances to mobile systems.

It should be noted that practical rendering systems often add a higher level of organization atop the BVH or k-d tree, similar to *scene graph libraries* in conventional rendering. One such organization is described by Wald et al. [WBS03], where the scene is divided into objects, which are organized in a top-level acceleration tree. Objects are then handled depending on their type of animation. Completely static objects can use a pre-built high-quality tree. Rigid-body motions are handled by translating and rotating incoming rays before traversing the object tree. Finally, objects with unstructured animation require full rebuilds on each frame.

This chapter surveys the state of the art related to the contributions of this Thesis on hardware BVH construction. First, we introduce basic methods used to evaluate the quality of BVHs, followed by a description of state-of-the-art fast BVH construction algorithms. Next, previous hardware builders are reviewed. Since the algorithm proposed in the Thesis work makes heavy use of sorting, we also discuss sorting accelerator architectures. Finally, the relation of the Thesis work to the state of the art is summarized.

2.1 BVH Construction Methods

Acceleration data structures can be said to have different levels of *quality* depending on how they are constructed. With a high-quality tree, ray traversal requires fewer intersection tests and traversal steps, and consequently, images are rendered faster. A



(a) *Bunny* model. (b) SAH sweep build, $C = 36.9$. (c) Binned SAH sweep build, $C = 39.0$. (d) LBVH build, $C = 44.5$.

Figure 2.2: Example of the effect of tree quality on traversal cost, ranging from a slow, high-quality build method (SAH sweep) to a fast, low-quality build method (LBVH). Cost is measured with primary ray tracing. Red color component denotes the number of inner node visits per pixel, blue denotes the number of ray-triangle tests.

standard way to evaluate the quality of a BVH tree is its Surface Area Heuristic (SAH) cost, introduced by Goldsmith and Salmon [GS87] and adapted to BVHs by Macdonald and Booth [MB90]. The SAH cost of a data structure is the expected cost to traverse a random non-terminating ray through the scene. For a BVH, the heuristic cost C can be computed as:

$$C = C_{node} \sum_{i=0}^{n_{\text{nodes}}} \frac{A(N_i)}{A(R)} + C_{leaf} \sum_{i=0}^{n_{\text{leafs}}} \frac{A(L_i)}{A(R)} + C_{tri} \sum_{i=0}^{n_{\text{leafs}}} \frac{P_i A(L_i)}{A(R)},$$

where $A(N_i)$ and $A(L_i)$ are the surface areas of the given inner nodes and leafs, respectively, $A(R)$ is the surface area of the scene AABB, P_i is the primitive count within a given leaf, C_{node} is the cost of traversing an inner node, C_{leaf} the cost of traversing a leaf, and C_{tri} the cost of a primitive intersection test [KA13].

In addition to measuring tree quality, SAH is widely used as a basis for tree build algorithms. The most well-known such algorithm is the *SAH sweep* [MB90] which builds BVHs by recursively partitioning the set of primitives top-down. At each step, SAH sweep attempts to split the remaining primitives into two subsets along all possible axis-aligned planes. The split plane giving the lowest SAH cost is selected and used to generate an inner node, the algorithm proceeds recursively into the two children, and recursion terminates when there is no split available which would improve the SAH. This algorithm is generally used as a gold standard against which other builders are benchmarked. Builders are often evaluated based on their resulting SAH cost normalized to a tree constructed with SAH sweep. A less expensive variant is the *binned SAH sweep* [Wal07], which considers only, e.g., 8 or 16 candidate split planes per axis, trading off quality for build speed.

Figure 2.2 shows an example with trees of different quality, and the resulting measured traversal cost in node visits and intersection tests. It is visible that cost is unevenly distributed over the image, and mostly concentrated on edges of objects, where rays may pass closely by many triangles without hitting them. In practice, traversal performance is also highly viewpoint-dependent.

It has been shown [AKL13] that SAH is an imperfect quality measure since, in practice, rays tend to begin and terminate inside the scene. The heuristic can be improved by introducing a term for AABB overlap. In practice, SAH-based top-down construction

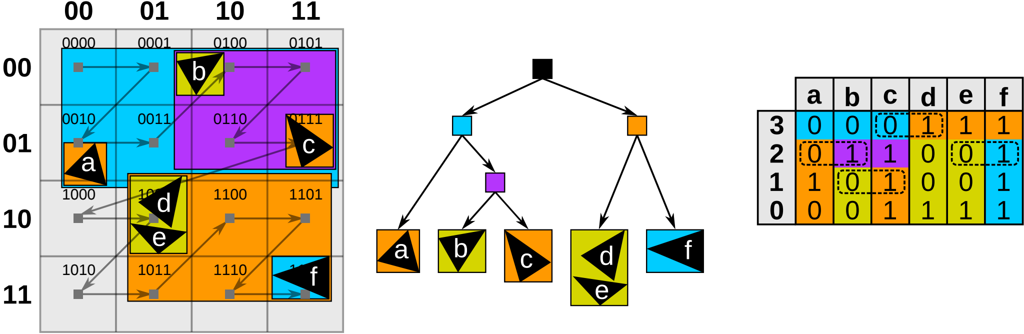


Figure 2.3: Example of LBVH. Left: locations of primitives in space, overlaid with a Morton code grid. Right: Morton codes of each bit. Center: generated hierarchy.

tends to also give good overlap, but some more complex build algorithms may further optimize SAH cost without giving a corresponding traversal performance benefit.

2.1.1 Linear BVH

Most of the recent fast GPGPU BVH builders aimed at real-time operation are based on the LBVH algorithm by Lauterbach [LGS*09], which organizes triangles by placing them on a space-filling curve. LBVH construction has roughly linear complexity in terms of arithmetic operations and memory accesses with respect to scene primitive count, compared to $\mathcal{O}(n \log n)$ for top-down SAH builds.

LBVH can be divided into four main stages:

- In *Morton code computation*, a regular $2^k \times 2^k \times 2^k$ grid is fitted in the scene AABB. All primitives are then assigned a Morton code by quantizing their AABB centroids to the grid and interleaving the bits of the grid coordinates. Typically, $k = 10$ (for 30-bit codes), or $k = 21$ (63-bit codes).
- Next, the primitives are *sorted* into Morton code order with a GPU-accelerated sorting algorithm such as a parallel prefix radix sort.
- In *hierarchy emission*, the bits of sorted Morton codes are used to generate the topology of the final BVH tree, i.e., generate child pointers for each node. In some implementations, primitives with identical Morton codes are grouped into the same leaf [LGS*09], while in others, an arbitrary hierarchy is generated so that there is one primitive per leaf [Kar12].
- Finally, *AABB computation* converts the BVH topology into a final BVH by computing node AABBs, in a step similar to BVH refitting.

LBVH is illustrated in Figure 2.3. Primitives are assigned Morton codes by binning to a coarse grid (left). The hierarchy (center) is generated based on bit patterns in sorted codes (right). For example, here the most significant bit corresponds to a halfway split of the scene along the vertical axis: primitives with a '0' bit are in the upper half of the scene volume, and primitives with '1' in the lower half. The root node splits primitives into subsets based on this bit. The subsets are recursively subdivided according to lower Morton code bits to form the rest of the hierarchy.

The original LBVH of Lauterbach already gave order-of-magnitude faster builds than previous algorithms, but has later been extensively optimized. Out of the above stages, Morton code computation is cheap and trivially parallel, and GPU sorting is a well studied general problem. Hence, most attention has gone to improving hierarchy emission and AABB computation.

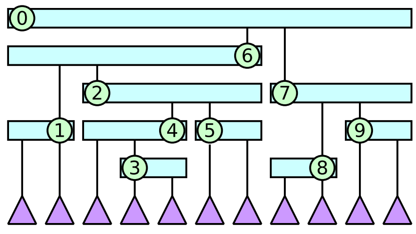
The main insight enabling parallel hierarchy emission is that each LBVH node corresponds to a continuous range of primitives with an identical Morton code prefix, i.e., the resulting topology is a *binary prefix tree*. For example, in Figure 2.3, the children of the root node have prefixes of 0 and 1. Moreover, a node can be considered to have a *hierarchy level* l, such that a node with low level has a longer prefix, and a *split position*, which is the primitive array index separating its child ranges. By comparing the Morton codes of successive sorted primitives, it is possible to determine whether that boundary is a node split, and determine the hierarchy level of that split. Lauterbach's method generates a per-primitive split list, and then sorts the splits by their level and their primitive range upper bound, resulting in a memory layout where it is easy to find parent-child relationships between nodes.

To understand the GPU implementation of the above algorithm, and later improvements, we need to examine the concept of *parallel prefix sum*, or *scan* [HSO07]. Scans are used in GPGPU computing to perform operations where each parallel thread of execution produces a dynamic amount of output data, which should be stored continuously. To parallelize such a problem, it is broken into multiple passes. The initial pass computes the amount of output for each work item n, o_n. An intermediate pass computes, for each item, a prefix sum of the output sizes of all earlier items: $p_n = o_1 + o_2 + ... + o_n - 1$. A final pass performs the computation, and places the outputs at memory locations $p_n ... p_n + o_n - 1$. A common application is the parallel prefix radix sort, which parallelizes the binning stages in radix sorting. Parallel prefix sums allow highly parallel computation of some problems at the cost of extra computation.

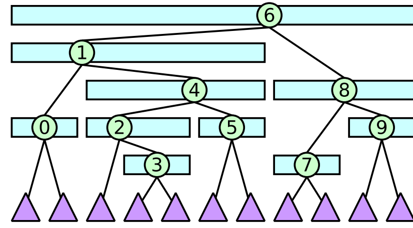
Lauterbach's LBVH first performs scans to generate the split list; then further scans for each hierarchy level to determine the memory locations of nodes. A drawback of the method is that scans are performed on intermediate data arrays much larger than n, and hierarchy emission generates many *singletons*, or nodes with only one child. Singletons are later removed with a postprocessing pass, but the leftover nodes are fragmented in memory, and construction requires a large memory arena. The later Hierarchical Linear Bounding Volume Hierarchy (HLBVH) of Pantoleoni and Luebke [PL10] addresses these drawbacks: it only emits full nodes and, furthermore, the number of kernel executions is reduced by processing p hierarchy levels at a time ($p = 3$ in the paper [PL10]). Each level set takes as inputs the leafs of the previous launch, which are further refined into treelets of up to $p^2 - 1$ nodes. For each level set, a single kernel launch first generates an array of treelet *block descriptors*, after which a scan is used to emit the corresponding hierarchy.

Garanzha et al. [GPM11] also perform hierarchy emission top-down, but do this in a single kernel call by means of the *task queue* mechanism in CUDA. Each task in the queue encodes the primitive range covered by an inner node. For each input range, the algorithm compares the first and last Morton codes in the range; determines the highest differing bit; and uses a binary search to find the corresponding split position. The resulting child ranges on each side of the split are then fed into the task queue. Initially, the kernel is fed a single task corresponding to the root node, which encompasses all primitives, and the algorithm then proceeds top-down.

Karras [Kar12] performs single-kernel hierarchy emission without task queues by defining



(a) Memory layout of Karras [Kar12]



(b) Memory layout of Apetrei [Ape14]

Figure 2.4: BVH memory layouts used to accelerate parallel GPGPU LBVH construction. The method of Karras (a) places each node at the edge of its primitive range closest to the parent split. The method of Apetrei (b) places nodes at their own split positions.

a special node layout, illustrated in Figure 2.4a, where the children of a node are placed at indices surrounding the node's split position. A GPU thread is launched for each inner node, which can determine whether it is at the left or right end of its primitive range by examining the surrounding Morton codes. The algorithm then finds the other end of the range by means of a binary search, and performs a second binary search in the range to determine the split position, which gives the child indices. AABB computation proceeds via a bottom-up reduction: each GPU primitive is assigned a GPU thread which traverses up the hierarchy, joining its AABB to the parent, via parent pointers generated earlier. Each inner node has an atomic counter which is used to kill the first thread entering the node, and allow the second thread to pass. This work also notes that the LBVH can be easily adapted to building point cloud kd-trees and octrees.

Apetrei [Ape14] performs joint hierarchy emission and AABB computation with a bottom-up reduction similar to the one used by Karras for AABB computation. Each GPU thread starts at a single primitive and works up the hierarchy, joining its primitive range and AABB to the parent. By using a memory layout where each node is at its split position, as shown in Figure 2.4b, the parent of any primitive range is found by examining the Morton codes at the ends of the range, without incurring a binary search. This results in a very simple implementation, and is currently considered the state of the art.

The above works are limited to scenes small enough to fit in GPU memory. Zeidan et al. [ZNA15] consider out-of-core LBVH construction, i.e., the case of very large scenes. In their framework, blocks of data are paged into GPU memory on demand. Primitive sorting is performed by means of an external sorting algorithm which first performs GPU-accelerated block sorts followed by a multi-way merge. The algorithm then builds a top-level hierarchy top-down, until it encounters nodes with primitive ranges small enough to fit in GPU memory, which are processed on GPU. An earlier example of out-of-core LBVH-like build is the method of Kontkanen et al. [KTO11] for point-based global illumination, which uses Morton code sorting to build an octree of points.

2.1.2 Refinement Based Construction

Plain LBVH, as described above, exhibits very fast build speeds but poor tree quality. Hence, a common approach in recent tree construction research has been to begin with a LBVH tree and postprocess it to improve quality. Doyle [DTM17] calls this approach *refinement based construction*.

The initial LBVH and HLBVH papers already propose refinement based techniques. Lauterbach et al. [LGS*09] generate a high-level hierarchy with LBVH, and then refine the large leaves with binned SAH sweeps. Pantoleoni and Luebke [PL10], in turn, propose to rebuild the upper levels of the hierarchy with binned SAH: the reasoning is that high levels of hierarchy have far more impact on tree quality per node and, therefore, the high-quality rebuild gives a better quality improvement per compute effort. Conceptually, in this kind of a *top-level build*, the high bits of the Morton code are used to bin the scene primitives into the cells of a coarse grid, and a LBVH subtree is built inside each gridcell: the subtree roots are then used as primitive inputs for a high-quality build.

The method of Garanzha et al. [GPBG11] performs the initial steps of LBVH, Morton code computation and sorting, but follows up with an approximate binned SAH sweep build: the SAHs of candidate splits are evaluated based on primitive counts recorded in a multi-level grid.

Bittner et al. [BHH13] give a generic BVH optimization algorithm based on removing inefficient subtrees and inserting them to more advantageous positions in the tree. The method is evaluated on a tree constructed with a SAH sweep, but could also be used to optimize LBVH trees.

Karras and Aila [KA13] have proposed a local optimization technique which rearranges small treelets of, e.g., 8 nodes to improve tree quality. Dynamic programming is used to find a treelet topology which maximizes SAH. The method, later dubbed Treelet Restructuring Bounding Volume Hierarchy (TRBVH), maps well to GPGPU computing as treelets are abundant and can be optimized in parallel. The approach is reminiscent of *tree rotations*, but considers a larger search space. The resulting trees are often superior to sweep SAH.

The Agglomerative Treelet Restructuring Bounding Volume Hierarchy (ATRBVH) method of Domingues and Pedrini [DP15] replaces the exhaustive search in TRBVH with an agglomerative rebuild, improving on the runtime of TRBVH at the cost of a small decrease in quality. Agglomerative construction considers all available pairs of nodes, and iteratively joins nodes with the lowest cost metric, in this case the surface area of the resulting node, until only the root node remains. Agglomerative construction is not among the fastest builders when applied to complete trees, having $\mathcal{O}(n^2)$ complexity with regard to the number of input primitives, but it is inexpensive for treelets of a few nodes.

Meister and Bittner [MB17] give a *parallel locally-ordered clustering* method which is a relative of LBVH − the method includes Morton code sorting and AABB computation, but hierarchy emission proceeds bottom-up by joining nearest-neighbor nodes, searched from a neighborhood on the Morton code order.

Hunt et al. [HMF07] proposed an approximate SAH sweep build which, instead of sweeping the complete primitive list to find optimal splits, samples a constant-sized *cut* of nodes from an initial low-quality hierarchy. The idea is recently applied on modern CPU-based BVH construction by Hendrich et al. [HMB17], using LBVH to generate the initial hierarchy.

2.1.3 Spatial Splits

Several methods have been proposed for splitting large triangles into multiple triangle references in BVH construction, giving, in effect, a hybrid structure between k-d trees and BVHs. This has little effect on scenes with evenly tessellated geometry, such as the

Stanford Bunny, but often gives a significant jump in ray tracing performance for complex scenes with unevenly sized primitives. The drawback is that the resulting hierarchy has a somewhat larger memory footprint due to the duplicate references, and cannot be used for refitting.

A seminal work in this direction was Split Bounding Volume Hierarchy (SBVH) by Stich et al. [SFD09], which often gives a ca. 25% traversal performance improvement over a SAH sweep. Their work, largely speaking, modifes a SAH sweep builder to consider spatial splits in addition to object splits, and selects the most advantageous split in terms of SAH. Recently, Ganestam and Doggett [GD16] propose a builder with triangle splits which gives quality similar to SBVH with a runtime closer to binned SAH. Out of the LBVH-based fast builders described above, TRBVH and Garanzha et al. [GPBG11] incorporate spatial splits generated with faster, lower-quality methods.

2.1.4 Other Build Algorithms

Beside Morton code sorting, some recent builders aim for fast, high-quality builds with other operating principles which deserve mention.

The gold-standard SAH sweep and SBVH algorithms build the tree top-down. An elegant way to apply similar ideas bottom-up is to start with a leaf list which initially contains all primitives, and iteratively merge the leaf pair with the lowest distance metric into an inner node, until only the root node is left. Typically, a SAH-like distance metric is used. This approach is called *agglomerative construction*. Naive agglomerative construction is $O(n^3)$ wrt. the number of input primitives and only suitable for trivial scenes, but is amenable to optimization. Walter et al. [WBKP08] use a low-quality kd-tree to accelerate the inner loop of the algorithm. Later work by Gue et al. [GHFB13] proposes Approximate Agglomerative Construction (AAC), where Morton code sorting is used to find an approximate spatial neighborhood for each leaf: the search for the closest primitive is limited to this neighborhood. A GPU builder by Meister and Bittner [MB17] works on a similar principle.

The Bonsai builder by Ganestam et al. [GBDAM15] is based on a fast implementation of SAH sweep which is optimized by means of multithreading, Single Instruction Multiple Data (SIMD) vectorization and algorithm-level improvements. The full algorithm uses a cheap top-down spatial median split to divide the scene into subsets, organizes each subset with SAH sweeps, and connects the resulting trees with a toplevel SAH sweep. Optionally, the subset trees are pruned to improve toplevel build quality. The resulting CPU build is only ca. 2× slower than the GPU TRBVH build [KA13], though with some generations newer hardware.

Meister and Bittner [MB16] have proposed a builder based on k-means clustering, which is more often seen in machine learning. The algorithm randomly selects bin prototypes from the primitive list, places each primitive in the bin of the closest prototype, and repeats with adjusted prototypes. A surface area based distance metric is used. The algorithm is based on recursive top-down partitioning like, e.g., binned SAH sweep, but generates multiple levels of hierarchy per pass, potentially allowing construction with fewer sweeps and less memory traffic.

2.1.5 Refitting

Aside from rebuilding the acceleration data structure from scratch on each frame, another strategy to handle animations is to *refit* the data structure to the new geometry, using the structure from the previous frame as a base. BVHs are particularly easy to refit by keeping the original tree topology and recomputing all AABBs [LYMT06, WBS07]. This is equivalent to the AABB computation stage of LBVH, i.e., a small sub-component of a very fast tree builder.

However, refitting has two drawbacks to offset the high build performance. Firstly, the quality of a BVH tends to deteriorate with successive refits. Two main strategies to improve quality are to *refresh* the tree with an asynchronous high-quality build [IWP07, WIP08], and to recover quality with postprocessing, similarly to refinement based construction techniques. Proposed postprocessing techniques include tree rotations [KIS*12], and restructuring based on culling detected empty spaces and overlaps [YL14]. Lauterbach et al. [LYMT06] give a heuristic criterion for initiating asynchronous rebuilds. Yoon et al. [YCM07] propose criteria to detect and rebuild low-quality subtrees.

The second drawback of refitting is that it is only applicable to mesh deformations. This is a fairly common class of animation, including, e.g., skeletal animations, but various procedural animation methods. E.g., physically simulated fluids are often rendered with *marching cubes* [LC87] or a similar isosurface method, where the amount and connectivity of triangles is dynamic. A more common difficult case in practical applications might be insertion and deletion of geometry.

2.1.6 Summary

In summary, BVH construction and update has been a subject of intensive research. It is hard to compare the many proposed methods since their results have been obtained with varying hardware platforms, input datasets and degrees of optimization. However, LBVH [LGS*09] with Karras' [Kar12] or Apetrei's [Ape14] algorithm is certainly the fastest published method to build a BVH from scratch. When aiming for a fairly high quality in addition to real-time build performance, the state of the art is likely one of the refinement based construction methods which build on LBVH. If animations are restricted to mesh deformations, refitting-based methods may give a better runtime-quality tradeoff.

2.2 Hardware Accelerated Construction

A large body of scientific work investigates hardware architectures for ray tracing accelerators, often for mobile systems. In a mobile environment, real time tree construction on programmable hardware is unlikely to be feasible for nontrivial scenes. In order to enable dynamic scenes, tree construction hardware is of high interest. Likewise, in a desktop environment, tree construction for a large scene can take up most of a GPU's resources, and an accelerator unit could free up the GPU for rendering.

Furthermore, hardware tree builders may have applications beyond ray tracing – Heinzle et al. [HGBG08] have proposed a processing unit for point sets, and identified tree construction as future work. In a similar vein, Raabe et al. [RHAZ06] have proposed an accelerator architecture for triangle mesh collision detection, which makes use of BVHs, and leaves deformable objects as an open problem.

2.2.1 Binned SAH Acceleration

The first hardware tree builder for ray tracing was proposed by Doyle et al. [DFM13], around the binned SAH sweep algorithm. This builder is denoted here as Binned SAH Accelerator (BSAHA).

Binned SAH is a recursive top-down build algorithm. At each recursion step, the builder considers a range of primitives, computes the SAH costs for S candidate splits along the three coordinate axes − typically $S = 8$ or $S = 16$ − and selects the split with the best cost. The algorithm then partitions the nodes on each side of the split to contiguous index ranges and proceeds recursively down to each child range. Recursion terminates when there is no split available that would improve the SAH cost compared to generating a leaf. Split SAHs are computed by first placing each primitive in one of $S + 1$ *bins* which each have an AABB and a primitive count − the SAH for a split is then computed by joining the AABBs and primitive counts of the bins on each side of the split.

BSAHA includes separate pipelines for partitioning, binning and SAH computation. Two optimizations are used to reduce off-chip memory traffic:

- Sufficiently small primitive ranges are processed on-chip and
- the results of partitioning are immediately reused for the binning steps of each child node, halving the amount of DRAM reads in high-level sweeps compared to a naive implementation.

As a result, the architecture generates ca. $2 - 3\times$ less traffic than a software build with the same algorithm. In followup work, the architecture was prototyped on Field Programmable Gate Array (FPGA) and applied on volumetric data in addition to triangle meshes [DTM17].

2.2.2 Refit Acceleration

Another approach to hardware-assisted tree update is to accelerate BVH refitting, i.e., updating node AABBs to match new geometry while reusing the tree topology from a previous frame. The HART hardware architecture by Nah et al. [NKP*15] incorporates a Geometry and Tree Update unit (GTU) for this task. In addition to updating node AABBs, the GTU contains hardware pipelines for generating the updated geometry by interpolation between keyframes, and packaging the updated geometry in TriAccel data structure [Wal04] for cheaper intersection tests. Refitting can be parallelized between multiple GTUs, in which case each unit is assigned a range in the primitive array, and a toplevel refit is done once all the parallel low-level refits finish.

In order to control the tree quality degradation due to refitting, the HART system incorporates asynchronous CPU rebuilds [IWP07, WIP08]. The system gives real time updates for large scenes, but it is noted that it depends on frame-to-frame coherence, and would have difficulty with, e.g., rapidly changing scenes and object insertion and deletion, and changes in mesh topology.

Earlier, the DRPU architecture by Woop et al. [WBS06] also included a programmable *update processor* for recomputing bounds in a Bounded k-dimensional tree (B-KD tree). A separate update processor program needs to be generated for each dynamic object. Programs can be designed to keep reused vertex coordinates in special *bound registers*, reducing memory traffic. There are no facilities to refresh degrading tree quality over the

course of animation, and it is emphasized that the system needs coherent motion to work well.

2.2.3 *k*-Dimensional Tree Accelerators

Apart from BVHs, accelerators have been proposed for constructing k-d trees, though these are computationally more complex to construct. The *RayCore* ray tracing architecture [NKK*14] incorporates a Tree Build Unit (TBU) to support dynamic geometry, with several high-level optimizations to make the problem more tractable:

- Separate trees are used to index dynamic and static geometry, and the static tree is built using a high-quality offline algorithm. Each ray first traverses the dynamic tree, followed by the static tree.
- The dynamic tree uses a two-level approach similar to Wald et al. [WBS03]: on each frame, a separate tree is built for each dynamic object, followed by the construction of a top-level tree organizing the objects.
- Each dynamic object tree is built in two levels: the high-level hierarchy is generated with a binned SAH build, while subtrees small enough to fit in on-chip working memory are built with a full, sorting-based SAH sweep. The reasoning is that using SAH sweep for the high-level hierarchy would give an unfavorable random memory access pattern. Separate hardware pipelines are allocated for binned and sweep SAH.

The RayCore TBU enables real-time animations with up to ca. 50K triangles of dynamic geometry, but the results underline the difficulty of k-d tree update. Even with the above algorithm-level optimizations, the builder is an order of magnitude slower than BSAHA [DFM13] while using a similar amount of logic − ca. 100 floating-point multipliers for RayCore TBU and 200 for BSAHA.

The later *FastTree* builder [LDNL15] is based purely on Morton code sorting, and consequently achieves faster builds than the TBU. In this approach, triangle references are generated for each Morton code gridcell overlapped by a triangle's AABB. References are sorted with a radix sort based on hardware parallel prefix sums. When emitting the hierarchy, spatial split planes are generated based on Morton codes.

A weakness of FastTree is that tree quality is not evaluated − as described, it generates triangle references for each Morton code gridcell overlapped by the triangle AABB, potentially leading to a very large number of extra nodes. Intuitively, it appears the quality penalty of Morton code based building would, then, be larger than with SAH builds. In a surprising result, a later software implementation of the same algorithm is reported to give higher traversal performance than a SAH build [LDG17] − however, the result is obtained with a complex methodology which adjusts runtimes measured from different GPUs, rather than a direct comparison.

2.2.4 Imagination Technologies Builder

Related to the recent Imagination Technologies ray tracing GPU project [Pow15], McCombe et al. [MDPN16] patented a hardware builder which processes a single stream of input primitives into, e.g., a BVH or a k-d tree. This is accomplished by binning the input primitives into voxels picked from several volumetric grids of different granularities.

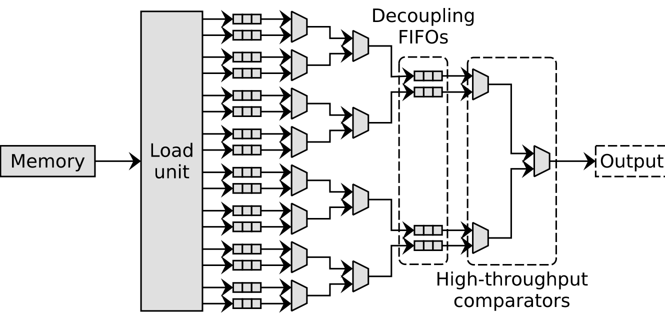


Figure 2.5: 16-way comparator tree used for multi-way merges in FPGA external sorting accelerators [KT11, CO14].

A *voxel cache* stores entries corresponding to voxels, which including primitive lists and compressed bounding volumes. When a voxel entry is evicted from the cache, it is processed into a part of an acceleration tree, and a reference to the resulting tree is added to a cached voxel of coarser granularity.

This streaming volumetric builder has several properties of interest: it is, e.g., able to take advantage of data compression during construction, and can break elongated triangles into multiple bounding volumes. Sadly, no public evaluation exists on the performance of this builder, or the quality of the resulting trees. Another interesting aspect is the proposed use in photon mapping.

2.3 Sorting Hardware

Since sorting is a major component of Morton-code based tree builds, it is interesting to look at the literature on sorting accelerators. This literature can be divided into accelerators for internal and external sorting, which are used, respectively, on arrays small enough to fit on-chip, and on arrays so large that they need to be stored in slow off-chip memory. The best hardware design for internal sorting also depends on the input size, e.g., very small arrays can be handled with *sorting networks* [Knu99] at a rate of one array per cycle, but these quickly explode in complexity. Since only trivial 3D scenes fit on-chip, tree construction has more use for an external sort. The recent work on external sorting accelerators focuses on FPGA-based designs built around the *multimergesort*, or *external mergesort* algorithm [Knu99].

Multimergesort first sorts *runs* of data which are small enough to fit in local memory, and then repeatedly merges multiple runs together, until a single run remains. More formally, the goal is to sort N data elements, taking advantage of M elements of fast local memory while $N >> M$, and minimizing traffic to a slow external memory. Moreover, the external memory should be accessed in block transfers of size B. Each merge, then, reads M/B runs and writes out a single run. At the time of the algorithm's invention, the fast memory was Dynamic Random Access Memory (DRAM) and the slow memory was, e.g., a rotary magnetic drive, but contemporary work instead minimizes traffic to the DRAM in favor of on-chip Static Random Access Memorys (SRAMs) blocks.

The initial local sorts in hardware multimergesort could be implemented with any internal

sorting algorithm, for example, bitonic sorts are recently popular. Starting with the design of Koch and Torrensen [KT11], multi-way merges are largely performed by means of *comparator trees*, which consists of *comparators* that push the lower of the input values to the output. An exception is the work by Zhang et al. [ZCP16], where an FPGA accelerates block sorts, and the multi-way merges are done on CPU. A 16-way comparator tree is shown in Figure 2.5. A K-way tree requires $\mathcal{O}(k \log k)$ comparators and First In First Outs (FIFOs), however, only $\mathcal{O}(\log k)$ of them are active at a time. One difficulty with a large tree is to determine backpressure propagation, i.e., which comparators should consume their inputs and which should stall, in one cycle. This can be resolved by inserting decoupling FIFOs between some tree levels [KT11].

Later work focuses on improving the throughput of comparator trees, as they are in the basic design limited to serial operation by the final comparator in the tree. FPGASort by Casper et al. [CO14] places high-throughput comparators near the top of the tree, which consume and produce multiple data items per cycle for a throughput of 6. Mashimo et al. [MVCK17] give comparators with a throughput of 8. Srivastava et al. [SCPC15] propose to replace the top levels of the tree with bitonic networks.

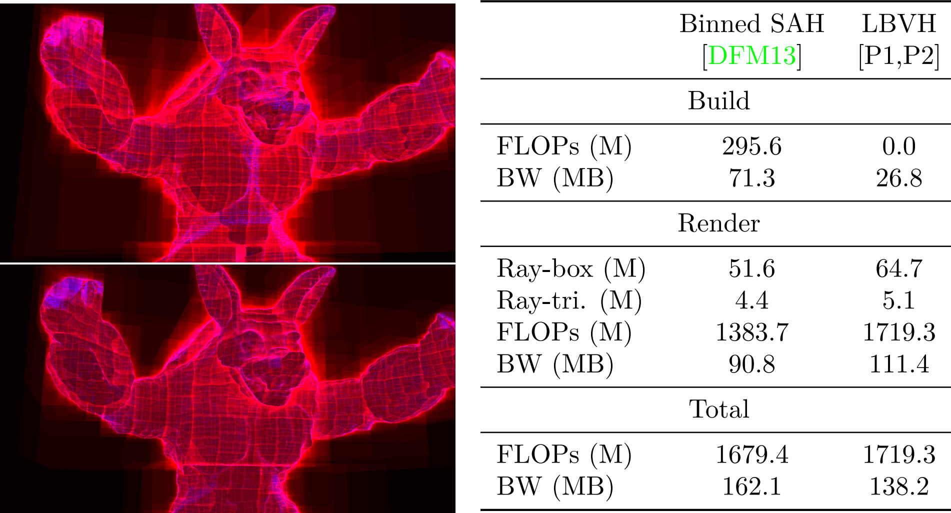
Since memory traffic is determined by the number of merge passes, it is desirable to build very large trees with, e.g., hundreds or thousands of sources. Large comparator trees may consume the resources of an entire FPGA, limiting the possible merge width. Usui et al. [UVCK16] propose a multi-way merge design with only a single physical comparator per tree level, in which the logical FIFOs are stored in Block Random Access Memorys (BRAMs). This gives logarithmic resource utilization in terms of merge width, except for BRAMs, and allows large merges: merge widths up to 4096 are evaluated.

In order to reach high operating frequencies on FPGA, sorter designs are creatively pipelined. Koch and Torrensen [KT11] pipeline comparators by comparing the more significant half of the on the first clock cycle, and the less significant half on the second. Casper et al. [CO14] also propose a complex pipelining strategy for the high-throughput comparators near the top of the tree. These techniques are a special feature of FPGA circuit design: Application Specific Integrated Circuit (ASIC) designs typically do not need this degree of pipelining to reach the desired operating frequency.

2.4 Thesis Contribution

To summarize the state of the art review above, the hardware tree updaters proposed to date are the refit-based HART architecture by Nah et al. [KSS*13], the tree builder by Doyle et al. [DFM13], which we refer to as BSAHA, and k-d tree builders [NKK*14, LDNL15]. In [P1,P2], we propose a new builder architecture based on LBVH, MergeTree. MergeTree applies the general traffic reduction approaches of streaming computation and local memory usage − used to good effect in BSAHA and TBU − to the computationally cheaper LBVH algorithm. Since primitive sorting is a major component of memory traffic, it is performed with an external sorting algorithm. The main novel contributions in [P1,P2] are:

- a streaming algorithm for HLBVH hierarchy emission and AABB computation in LBVH and
- an external sorting implementation based on a hardware heap [IK07] rather than a comparator tree.



	Binned SAH [DFM13]	LBVH [P1,P2]
Build		
FLOPs (M)	295.6	0.0
BW (MB)	71.3	26.8
Render		
Ray-box (M)	51.6	64.7
Ray-tri. (M)	4.4	5.1
FLOPs (M)	1383.7	1719.3
BW (MB)	90.8	111.4
Total		
FLOPs (M)	1679.4	1719.3
BW (MB)	162.1	138.2

(a) Traversal cost with LBVH (top) and binned SAH (bottom) builds.

(b) Memory layout of Apetrei [Ape14]

Figure 2.6: Detailed example comparison of MergeTree [P1,P2] to BSAHA [DFM13] in platform-independent metrics for one test scene. The Armadillo scene (213K triangles) is drawn with 1spp, one diffuse bounce, without next event estimation. BW: external memory bandwidth, FLOPs: floating-point operations. Addition, subtraction and multiplication are counted as a single FLOP, reciprocal as 3. Tree build BW is exact; rendering BW depends heavily on the cache hierarchy. Here, the simple cache model from [P3] is used. The Armadillo model is courtesy of the Stanford Computer Graphics Laboratory.

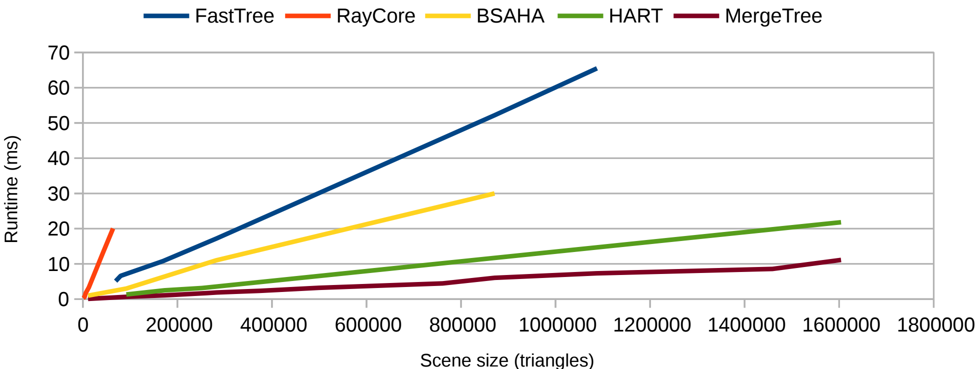


Figure 2.7: Reported runtimes of hardware tree builders.

The combination of external sorting and streaming allows MergeTree to use ca. 3× less bandwidth than software LBVH, while the heap-based sorter gives a compact silicon area. A similarly economical sorter architecture was later proposed by Usui et al. [UVCK16].

The proposed architecture gives distinct advantages compared to the state of the art: Compared to BSAHA, MergeTree uses an inexpensive algorithm, allowing significant reductions in build time, energy, silicon area and memory traffic. BSAHA, e.g., comprises

hundreds of Floating Point Units (FPUs) while MergeTree performs no floating-point arithmetic. As a drawback, the quality of output trees is lower, increasing the cost of ray tracing. A part of the quality can be recovered via *toplevel builds*, which are inexpensive enough to run in real time on a mobile CPU.

The tradeoff between the builders depends heavily on scene complexity and rendering parameters, and is quantified in [P2] for a variety of scenes and parameters. A worked example using platform-independent metrics is shown for a single test scene in Figure 2.6. In this case, the low quality of LBVH significantly increases the arithmetic and bandwidth cost of traversal, but these costs are compensated by savings in the tree build. Larger scenes tilt the tradeoff in favor of LBVH, and more complex visual effects, in favor of BSAHA. It should be noted that both hardware units use ca. 3× less bandwidth than a corresponding software builder.

Compared to the Morton code based k-d tree units, MergeTree contributes memory traffic optimizations for sorting. Combined with the simpler hierarchy emission of BVHs, this gives a much faster build speed than seen in the k-d tree units.

Refitting hardware such as HART is likely cheaper and faster than MergeTree to some degree, though not as much as software refitting is when compared to tree construction. A rough comparison is included in [P3]. However, refitting can only handle a limited class of animations, i.e., mesh deformations such as skeletal animation. Interesting classes of animation are out of reach, such as fracturing and fluid simulations rendered with marching cubes. Moreover, a refitting system requires full builds whenever new geometry is introduced, and at intervals to refresh the degrading tree quality.

A summary of reported build speeds of MergeTree and related work is shown in Figure 2.7. It is visible that MergeTree gives the fastest build performance out of hardware builders in the literature.

As with the related software algorithms, the system-level tradeoff between slow, high-quality BSAHA and fast, low-quality MergeTree depends heavily on the scene, viewpoint and visual effects in use. In order to obtain at least a rough estimate of the tradeoff, in [P2] we modeled the energy consumption of a system where a tree builder first organizes the scene, and a hardware ray tracer uses the resulting tree to render an image. Given a modest amount of incoherent secondary rays, MergeTree becomes preferable around 200000 animated triangles. It should be noted that in large scenes, a high-quality build takes up most of a mobile power envelope. In other words, rendering sufficiently complex lighting effects with many secondary rays would make high-quality builds preferable again in large scenes, but doing this in real time would run into the thermal dissipation limits of a mobile device.

3 Rebuilding and Refitting Compressed BVHs

Ray tracing is a highly memory-intensive process, and especially when using a specialized accelerator, the memory hierarchy easily becomes a performance bottleneck and major driver of energy consumption [KSS*15]. A BVH node has a similar memory footprint as a primitive, but a typical ray traversal touches $1-2$ orders of magnitude more nodes than primitives. Hence, node accesses account for most of the stress on each level of the memory hierarchy. As a result, a wide variety of BVH variants has been proposed that attempt to encode the inner nodes of the hierarchy more compactly, in order to reduce memory traffic and improve cache hit rates.

BVH compression is challenging since ray traversal accesses the data structure in an unpredictable order, performing only a small amount of computation per access. In addition to being compact, a CBVH needs to support random accesses with a low decompression overhead per traversed node. Various general tree compression methods have been proposed [KM90] which do not support random access, and are thus ruled out. Moreover, compression schemes often trade off some of the uncompressed BVH's performance at pruning intersection tests − as a result, some schemes achieve extreme compression rates, but the measured memory traffic during traversal actually increases.

The performance effect of compression depends on the behavior of the cache hierarchy. For example, $2\times$ data compression doubles the effective size of caches, but the resulting effect on performance and memory traffic depends on the specifics of the application and the memory hierarchy. A rough heuristic is that doubling cache size reduces the miss rate by a factor of $\sqrt{2}$ [HSPE06]. If compressed tree nodes are small compared to cache line size, memory accesses may end up fetching a large amount of data which is never referenced [Smi87]. Several works work around this by introducing cache-optimized tree layouts where nodes on the same cache line are likely to be accessed by the same ray-thread [YM06, NPK*10, LV16].

Aside from performance optimization in conventional ray tracing, BVH compression has also been motivated by rendering of very large scenes, where the challenge is to fit the complete scene in main memory, or perform *out-of-core rendering* where parts of the scene are paged in to fast memory on demand. This type of system tends to aim for maximum compression ratio for all data used in traversal, including primitives [LYM07, LYTM08, KMKY10, KBK*10, SE10], while compression systems aimed at smaller *in-core* scenes focus on node data.

Practically all research on CBVHs so far focuses on traversal performance in static scenes, and leaves aside questions of tree update. Recently CBVH based on *incremental encoding* is emerging as a key optimization in ray tracing accelerators [Kee14, VAMS16], which are

proposed for real-time rendering use, and it is, therefore, interesting whether compression can be made fast enough to support animated scenes. This Thesis aims to address the real-time update of incremental encoding BVHs. This chapter first gives an overview of BVH variants with reduced memory footprint, and then a more detailed introduction to the incremental encoding class of CBVH. The final section lays out how the present work relates to the state of the art in CBVHs.

3.1 BVH Compression Methods

As discussed above, compression systems aimed at in-core rendering tend to focus on inner nodes, which account for a large share of memory accesses. The vanilla BVH node includes an AABB with six 4B floating-point bounds and two child pointers, for a total size of 32B. Alternately, each inner node can include the AABBs of its children, for a node size of 64B. Since AABB bounds take up $\frac{3}{4}$ of each node, they are the focus of AABB compression methods. Some compression methods reduce the coordinate footprint enough that it also becomes interesting to compress pointer data.

3.1.1 Coordinate Compression

Some coordinate compression methods are compared via visual example in Figure 3.1. Most proposed methods are lossy, such that the decompressed bounding volume is larger than original, leading to false-positives in intersection tests, which cause additional traversal steps. We refer to this extra volume as *volume overhead*.

3.1.1.1 Shallow trees

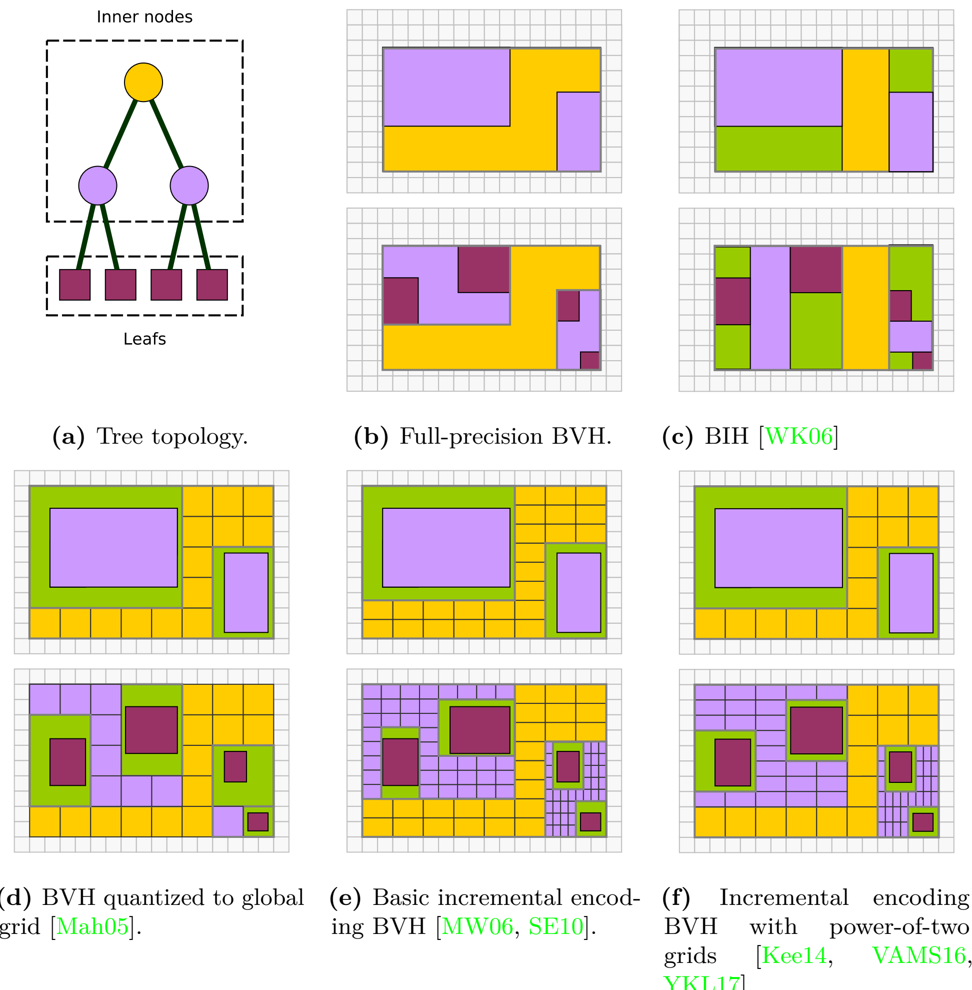
A popular variation on BVH is the MBVH, i.e., a BVH with a high branching factor, typically 4 or 8 children per node [WBB08, DHK08, EG08]. The main motivation for MBVHs is to take advantage of SIMD instructions in modern CPUs by performing multiple ray-AABB intersection tests in parallel. However, MBVH also has the effect of compressing at least the high levels of the hierarchy that are visited most often. E.g., a single 4-way MBVH node can replace a treelet of 3 binary nodes, while only using as much memory as two nodes. There is an extensive literature on MBVH, which is discussed in more detail in Section 4.2.

3.1.1.2 Plane sharing

Another lossless compressed BVH is the Shared Plane Bounding Volume Hierarchy (SPBVH) proposed by Fabianowski and Dingliana [FD09] as well as Ernst and Woop [EW11], which exploits the property that each coordinate of a BVH node AABB recurs again in one of its child AABBs. SPBVH adds bits to indicate whether each coordinate is encoded in the child or reused from the parent, reducing encoded coordinates by 50%. The drawbacks are that some control-oriented code is needed to decode the full AABBs, and each traversal stack entry needs to store a parent AABB in addition to a node pointer.

3.1.1.3 Bounding Interval Hierarchies

The BIH [WK06] also reuses parent coordinate bounds like SPBVH, but goes further by only encoding one bounding plane per AABB. Similarly to MBVH, similar data structures were independently proposed for ray tracing by several research groups as



(a) Tree topology.

(b) Full-precision BVH.

(c) BIH [WK06]

(d) BVH quantized to global grid [Mah05].

(e) Basic incremental encoding BVH [MW06, SE10].

(f) Incremental encoding BVH with power-of-two grids [Kee14, VAMS16, YKL17].

Figure 3.1: Comparison of basic BVH and some variants with compressed coordinate data. BIH and global grid BVH incur significant volume overhead (green), the grid especially in the lower levels of the hierarchy. Incremental encoding gives tighter bounds. Power-of-two grids simplify traversal at cost of slightly looser bounds. Note that 3-bit coordinates are used for grids and incremental encoding for clarity, but typical implementations use at least 5 bits.

Dual Extent trees (DE-trees) [ZU06], Spatial K-Dimensional trees (sk-d trees) [HHS06] and B-KD trees [WMS06]. Possibly the smallest proposed tree variant is the Minimal Bounding Volume Hierarchy (MVH) by Bauszat et al. [BEM10], which is similar to BIH, but further restricts the child AABBs to four possible truncations of the parent AABB, using two coordinate bits per node. A drawback of BIH and variants is high volume overhead due to empty space in nodes.

3.1.1.4 Grid Quantization

A straightforward way to reduce memory footprint of coordinates is store them as low precision integers instead of single-precision floating-point numbers, by snapping them to

a grid. Visual artifacts are avoided as long as the quantized bounding boxes completely envelop the original boxes. Since the compressed and decompressed boxes are larger than original, there is an overhead of additional intersection tests from false positives, similar to BIH. Also, the method incurs a decompression overhead from mapping the quantized coordinates back to the global frame. Global grid quantization was first proposed by Mahovsky [Mah05], who handles decompression overhead with custom hardware fixed-point traversal, and by integrating the decompression code to a packet tracer. They find that coordinate precision can be reduced to ca. 16 bits for simple models and 20 − 24 bits for complex scenes, before there is a major increase in intersection tests. Fixed-point grid traversal was also later studied by Hanika and Keller [HK07] and Heinly et al. [HRB*09], who also perform ray-triangle tests in fixed point. Hwang et al. [HLS*15] propose a mixed AABB encoding with 17-bit fixed-point lower bounds and 13-bit floating-point widths. Koskela et al. [KVJT16] show memory traffic and power savings with half-precision floating-point numbers.

3.1.1.5 Incremental Encoding

To recap, basic grid quantization can only compress coordinates by ca. 25% . . . 50%, depending on the scene, before the rough precision starts to significantly slow down ray tracing. A higher precision can be obtained by *hierarchical* or *incremental* encoding, where coordinates are encoded relative to the parent AABB. This approach was first proposed by Mahovsky et al. [MW06], who evaluate 4- and 8-bit encodings. 8-bit encoding is practically lossless and 4-bit encoding is still significantly less lossy than a 16-bit global grid. Recently, incremental encoding variants have been proposed that are efficient to traverse with custom hardware [Kee14, VAMS16], and a variant by Ylitie et al. [YKL17] gives state-of-the-art performance for software ray tracing on GPU. Incremental encoding is described in more detail in Section 3.2.

3.1.1.6 Two-Level Encoding

Several BVH variants with high decompression or volume overhead opt to use two-level encoding, where a high-level part of the tree is a basic BVH whose leafs are compressed hierarchies. E.g., Cline et al. [CSE06] use simple grid quantization for the second level of the hierarchy. The MVH [BEM10] and Implicit Object Space Partition (IOSP) [EBGM12] methods also evaluate two-level variants. Since high-level nodes are accessed more frequently, this allows the hybrid method to retain some of the ray tracing performance of simple BVH while keeping a high compression ratio.

3.1.2 Pointer Compression

Given a high degree of AABB coordinate compression, uncompressed pointers start to dominate the node size. Hence, coordinate compression is usually accompanied with some form of compressed pointer encoding.

3.1.2.1 Single-Pointer Layouts

In k-d trees, a popular depth-first layout [PJH16] places the left child directly after a parent node. Liktor and Vaidyanathan [LV16] evaluate this layout for BVHs. Ylitie et al. [YKL17] extend the concept to 8-wide compressed MBVHs by storing the multiple children of each node continuously in memory. The node, then, stores two full-precision base pointers for inner nodes and leaf children, respectively, and low-precision memory

offsets for child. This also allows a compact traversal stack with a single entry per group of intersected children.

3.1.2.2 Short Pointers

One general approach is to cluster parts of the tree together in memory, and use short pointers within each cluster. Hierarchical Mesh Quantization (HMQ) [SE10] use this scheme and handle links between clusters by inserting *forward nodes* with full-precision pointers. Liktor et al. [LV16] use a similar layout with *cache clusters* and *glue nodes*, and propose a cache-aware algorithm to divide compressed nodes into clusters.

3.1.2.3 Heap-like Layouts

Cline et al. [CSE06] propose a heap-like node layout where the memory locations of children are implicit from the parent node's location: given a tree of branching factor K, the children of a node with index i are stored at indices $Ki+1$ through $Ki+K$, and the parent is at $\lfloor \frac{i-1}{K} \rfloor$. As a result, storage of pointers is avoided entirely. The same approach is followed by, e.g., Bauszat et al. [BEM10] with successively smaller node coordinate encodings. A drawback is that the tree needs to be at least somewhat balanced to keep a reasonable memory footprint. Conventional high-quality tree builders do not make any attempt to balance trees, and a balancing algorithm is likely to sacrifice a significant amount of quality.

3.1.2.4 Implicit Hierarchy

The IOSP by Eisemann et al. [EBGM12] uses a heap layout to make the tree structure completely implicit in the order of the primitive array: the bounding volumes of a BIH-like tree are computed on the fly during traversal by examining primitives at specific indices. This does not appear to be an effective compression scheme for the purposes of reducing memory traffic, but does completely eliminate the memory footprint of BVH nodes.

3.1.3 Primitive Compression

When applying node compression, primitives quickly start to dominate the complete memory footprint, and their share of memory accesses may become significant. Hence, several compression schemes − especially those aiming at out-of-core rendering − address primitive compression.

In contrast to node compression, primitives are less tolerant to low-precision representation, as quantization could cause visual artifacts, especially with primary rays. Consequently, most compression schemes are lossless, and focus on eliminating redundant copies of coordinates. Interestingly, complete accuracy might not be needed in secondary rays when computing, e.g., indirect illumination [YCK*09]. Intersection tests for such rays could use approximate geometry and still give acceptable visual quality.

Some basic design choices can have a major effect on primitive footprint. For instance, triangles are often stored with precomputed intermediate values to accelerate intersection tests with, e.g., Wald's TriAccel method [Wal04] or Shevtsov's [SSKN07] or Havel's [HH10] methods. The resulting primitive is larger than simply storing the original nine coordinate values consecutively. This, in turn, consumes far more memory than storing each vertex once, and storing faces as lists of indices to a vertex array.

3.1.3.1 Triangle Strips

A popular way to represent triangle meshes compactly is to divide them into *triangle strips*, i.e., arrays of vertices $v_1...v_n$ that encode triangles $v_1v_2v_3, v_2v_3v_4..v_{n-2}v_{n-1}v_n$. The RayStrips ray tracing system [LYM07] uses the open-source Stripe package [ESV96] to divide the scene into strips, and builds a BIH-like tree hierarchy in each strip to reduce ray-triangle tests. The ReduceM system [LYTM08] develops on RayStrips by extracting strips with a SAH-based algorithm optimized for ray tracing performance, and optimizing the strip tree. Galin et al. [GA05] propose to use triangle fans instead of strips. Fans allow some optimization of the Möller-Trombore intersection test, but typically limit the amount of triangles per primitive. In addition to quantization and variable-length coding, Segovia and Ernst [SE10] encode their primitives as sets of strips. Each primitive begins with a code which identifies how the following vertices are divided into strips and singleton triangles.

3.1.3.2 Displacement Maps

The RayCore hardware unit has support for heightmap primitives based on *dynamical displacement mapping*. Novák et al. [ND12] propose the Rasterized Bounding Volume Hierarchy (RBVH) which approximates flat patches of the original geometry with heightmaps. The heightmap resolution can be adjusted to provide varying Level of Detail (LoD).

3.1.3.3 Subdivision Surface Caching

Smooth surfaces in 3D models are often created by specifying *subdivision surfaces*, where a triangulated surface is subdivided according to a mathematical model, for example, to approximate a Non Uniform Rational Basis Spline (NURBS) surface. Subdivision surfaces are a standard primitive in Computer Aided Design (CAD) tools [Ma05] and a staple in high-quality rendering for feature films [DKT98]. Some rendering systems are given the spline parameters, and subdivide each surface on demand so as to reach the desired visual quality, e.g., caching the subdivided geometry for reuse between rays [BWN*15, ÁWBW16]. This can be loosely regarded as a compression technique, since it allows rendering of scenes whose full triangulated geometry would be impractical to fit in memory.

3.1.3.4 Subdivision Surface Trees

BVHs variants are designed specifically to compress fully triangulated subdivision surfaces. Selgrad et al. [SLM*16] compress triangulated subdivision surface models by storing flat patches of geometry in a special quantized representation, which is conceptually somewhat similar to RBVH. Du et al. [DKY16] propose a BVHs with *tetrahedron swept sphere* bounding volumes, which give a compact representation and better triangle culling performance than AABB volumes. Due to the regular structure of subdivision surfaces, these methods give high compression ratios.

3.1.3.5 Level of Detail

Closely related to lossy geometry compression are *mesh simplification* and LoD techniques. In mesh simplification, a 3D model is represented in fewer triangles while preserving the original appearance. In LoD this is done adaptively, such that faraway objects are simplified more. There is a rich literature of LoD techniques, a comprehensive review of

which is given by Luebke et al. [Lue03]. Afrá et al. [Áfr12] propose a k-d tree for large models with built-in LoD. The tree contains special *LoD nodes* which store an average color and normal sampled over the underlying geometry − when viewed from far away, a LoD node is shown as a voxel of this approximate color, and when zooming in, the underlying geometry is asynchronously loaded to memory.

3.1.4 Entropy Coding

A basic approach in the field of data compression is *entropy coding* where recurring patterns of data are represented with fewer bits than rare patterns, by means of, e.g., Huffman coding [Huf52]. The Random Accessible Compressed Bounding Volume Hierarchy (RACBVH) system [KMKY10] combines several of the above techniques with entropy coding for out-of-core ray tracing. The basic concept is to split the input tree into large clusters of, e.g., 512..4K nodes, each of which is compressed as a separate block via dictionary-based entropy coding. During ray tracing, the compressed hierarchy is stored in slow external memory, and clusters are paged in and decompressed on demand. Both coordinate and pointer data is preprocessed by *predicting* values based on the previously encoded part of each block, and encoding delta values relative to the prediction, thus improving the compression ratio. For example, each node is predicted to be evenly split into two child AABBs along the longest axis. In addition, coordinate data is quantized to a global grid.

The Hierarchical Culling oriented Compact Mesh (HCCMesh) system [KBK*10] increases the compression ratio further by jointly compressing the primitive data with nodes and leafs. This allows the compressor to, e.g., eliminate coordinate bounds that are redundantly encoded in both primitives and their bounding boxes. Later work describes a GPU-accelerated rendering framework [KSY14] which decompresses HCCMeshs blocks when offloading them to the GPU.

3.1.5 Comparison

Given the volume of work and variety of approaches on BVH compression, it is interesting to compare compression methods quantitatively. Table 3.1 collects the reported compression ratios and intersection test overheads of several BVH variants compared to a plain BVH baseline. The comparison criteria are chosen to act as proxies of memory bandwidth usage: relatively few methods report bandwidth, and these depend on the particular machine. The rationale is that the compression ratio is a rough indicator of reduced memory traffic per traversal step, while increased traversal steps in turn increase traffic. Several interesting methods, e.g., [CSE06, SE10, LYTM08] did not report sufficient data to be included in the comparison − these tend to include primitive compression, which complicates comparison.

Methods based on BIH and pointerless heap layouts [BEM10, EBGM12] can produce very small data structures, but tend to have weak triangle-culling performance compared to plain BVH. This is partly alleviated by a two-level hierarchy with the compressed method as the lower level, but there is still an order-of-magnitude triangle test overhead.

Snapping AABB coordinates to a global grid is a low-hanging fruit, but triangle-culling performance starts to degrade sharply at ca. 50% compression. Incremental encoding reaches higher compression ratios. State-of-the-art software [YKL17] and hardware [LV16] CBVHs ray tracers combine incremental encoding with lossless MBVH and SPBVH,

Table 3.1: Comparison of some compact BVH variants. *Node visits, Triangle tests*: amount of node visits and ray-triangle intersection tests normalized to binary BVH ray tracing. *Node ratio*: compression ratio of inner node data in a tree, i.e., excluding triangle data. Note that the results have been obtained with different test scenes and are only roughly comparable. Given a choice, results with incoherent secondary rays were selected. MVH results are with single-ray tracing, but [BEM10] also show better results with packet tracing.

Method	Description	Node visits	Triangle tests	Node ratio
[EG08]	MBVH	28%[a b]	83%	1.2
[FD09]	SPBVH	100%	100%	2.0
[BEM10]	MVH	16254%	16784%	88.0
	Two-level MVH	481%	1460%	38.1
[EBGM12]	IOSP-4	127%	1139%	12.3
	IOSP-0	373%	3334%	∞
	Two-level IOSP	156%	548%	92.2
[Mah05]	Grid 24-bit	100%	100%	1.2
	Grid 20-bit	100%	104%	1.4
	Grid 16-bit	108%	171%	1.6
[HLS*15]	Grid 17-bit + 13-bit	104%	115%	1.7
[MW06]	Incremental 8-bit	101%	101%	2.7
	Incremental 4-bit	118%	118%	4.0
[Kee14]	Incremental 5-bit	156%[c]		4.0
[LV16]	Incremental 6-bit, SPBVH	117%	127%	8.0
[YKL17]	Incremental 8-bit, MBVH	39%[a]	76%	2.3
[KMKY10]	Entropy coding	101%[d]		9.5

[a] MBVH node traversals perform more ray-box tests than BVH nodes.
[b] The point of comparison is a BVH with one AABB per node.
[c] Reported traversal steps were not differentiated between nodes and triangles.
[d] Unclear whether reported intersection test ratio is for triangles, boxes or both.

respectively. Entropy coding [KMKY10] gives an even better ratio-overhead tradeoff, however, it is only randomly accessible at the granularity of large blocks.

3.2 Incremental Encoding

In the previous Section we found that incremental encoding is a good candidate for state of the art in BVH compression—it gives a good compression ratio while incurring relatively low volume overhead. This Section examines the work on incremental encoding in more detail.

The basic idea in incremental encoding is to set up a local coordinate grid in each inner node—which is contained inside the node AABB—and encode child AABB bounds in this grid. For example, Figure 3.2 shows one possible encoding used by Vaidyanathan et al. [VAMS16]. Here, the lower-bound and upper-bound coordinates of an AABB are encoded as offsets r_i, s_i in a local grid relative to the parent lower and upper bound, respectively. The size of a grid cell is set as the smallest power-of-two number where the parent is less than 2^b gridcells wide. Each axis has a separate grid size.

Mahovsky [MW06] implement b-bit encoding simply by fitting 2^b grid lines along each axis in each node AABB. Segovia and Ernst [SE10] use a similar scheme enhanced with compressed primitives and short-pointer encoding. Newer methods [Kee14, VAMS16, YKL17] restrict the grid to power-of-two sizes, avoiding expensive floating-point divisions.

Incremental encoding was revisited by Keely [Kee14] from the perspective of hardware acceleration. This work sketches a hybrid GPU architecture adding custom ray traversal

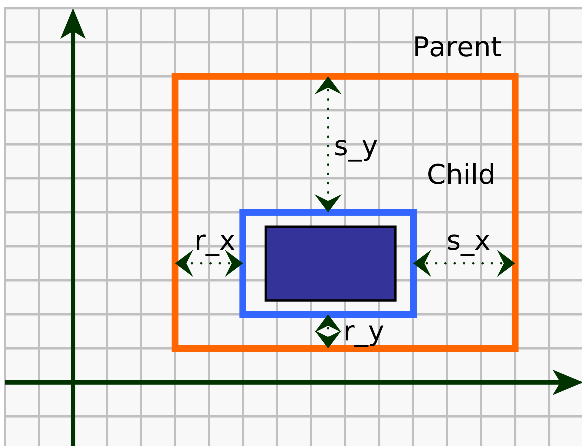


Figure 3.2: Incremental encoding example.

hardware to a conventional GPU, and incorporating incremental encoding, primitive compression and treelet scheduling to the traversal units. One way in which Keely's BVH variant is hardware-friendly is that grids are restricted to power-of-two sizes − this is advantageous since divisions and multiplications by a power-of-two number can be done cheaply in hardware by manipulating the floating-point exponent.

The main contribution by Keely [Kee14] is a method to perform ray-AABB intersection tests with low-precision arithmetic units, allowing traversal with less silicon area and power than full-precision. It was shown earlier that global grid encoding permits cheap hardware traversal [Mah05], but it was nontrivial whether this is the case in incremental encoding. Keely [Kee14] does this by means of *traversal point update*, where the origin of the ray is moved forward so that it can be quantized to the same local grid as the target AABB. As a result, the expensive floating-point multiplications of a ray-box test can be avoided.

Vaidyanathan et al. [VAMS16] give a more mathematically rigorous analysis of hardware-oriented incremental encoding, and proposes another approach to reduced-precision traversal, based on incrementally updating intermediate values on the *slabs* ray-box test during traversal. This is estimated to give more hardware savings than traversal point update, and guarantee watertight traversal. In addition, SPBVH encoding is used to improve the compression ratio. As a drawback, [VAMS16] requires a very large stack element compared to [Kee14]. Liktor et al. [LV16] integrate the same approach with short-pointer encoding and a cache-optimized memory layout, and propose a hardware architecture for a complete accelerator.

Recently, Ylitie et al. [YKL17] apply incremental encoding to software rendering on GPU, and achieve state-of-the-art traversal performance. This appears difficult at first sight due to the heavy per-node decompression overhead of software incremental encoding. In this work, most of the overhead is avoided by storing in each node the basis and scale of that node's local grid. In a binary node this would negate the compression advantage, so

a 8-wide MBVH is used to amortize the stored grid data over multiple child AABBs.

So far, the works on incremental encoding BVH have not aimed for real-time tree construction. The methods of Keely [Kee14] and Vaidyanathan [VAMS16] are evaluated in static scenes using trees built with the slow and high-quality SBVH algorithm [SFD09]. The method of Ylitie et al. [YKL17] combines SBVH with a novel tree collapsing to generate a shallow hierarchy, with a runtime measured in minutes.

3.3 Thesis Contribution

A wide variety of techniques has been proposed to compress ray tracing acceleration data structures, giving a variety of tradeoffs between compactness and overhead. The recently proposed incremental encoding CBVH appears to be a good contender for the state-of-the-art acceleration data structure in hardware accelerators. E.g., Keely [Kee14] estimates that a ray tracing GPU with incremental encoding could give orders of magnitude more ray traversal performance than current software state of the art, which would be sufficient for real time path tracing.

However, the literature so far has not addressed fast tree updates, so it is not certain that these benefits can be realized except when viewing completely static scenes. [P3] contributes the first treatment of real-time incremental encoding BVH construction in the literature, from the perspective of hardware acceleration.

A main finding in [P3] is that *streaming compression* is highly beneficial, i.e., a tree update that directly outputs a compressed tree is much cheaper, in terms of memory traffic, than an update which produces an uncompressed tree and is followed by a separate compression step. Streaming compression is straightforward for top-down compression methods, but nontrivial for bottom-up methods, as the encoding of a node depends on its parent. The main contribution of article [P3] is a streaming, bottom-up compression algorithm based on context estimation combined with backtracking to re-encode nodes whose estimated contexts are found to be invalid. Algorithmic techniques are proposed to reduce backtracking.

The proposed method is useful for updating incremental encoding BVHs up to Keely's method [Kee14]. A limitation and potential area of future work is that compressed trees based on SPBVH [VAMS16, LV16] cannot yet be compressed. Another interesting related work is the wide CBVH of Ylitie et al. [YKL17], which entirely bypasses difficulties with bottom-up compression by encoding in each node the context required to decode it. However, the method gives a limited compression ratio compared to hardware-oriented methods, and requires a time-consuming tree collapsing step for the best performance. It seems likely that a hardware ray tracer will opt to infer node contexts during traversal, in which case the proposed method is relevant.

4 Hardware-Accelerated Shallow BVHs

The previous Chapters discussed methods of constructing and updating BVH trees, which are used to accelerate ray traversal. The focus of this Chapter is on the implementation of ray traversal itself. Ray traversal has been used as a workload in practically all commodity computing hardware, including abundant CPU and GPU implementations, and more exotic platforms such as the IBM Cell processor [BWSF06] and Intel's Many Integrated Cores (MIC) platform [BWW*12].

The ray traversal task can be described as embarrassingly parallel−Even the primary ray tracing of a picture at 1080p resolution can be split into ca. 2 million threads of execution which can progress independently. The focus of traversal implementations is on leveraging the parallel hardware resources in a given hardware architecture. In the past decade, the amount of available parallelism has increased sharply and, consequently, traversal performance has followed suit.

Modern CPUs have two main mechanisms for parallelism: multi-core operation with Simultaneous Multi-Threading (SMT), and SIMD vector operations. It is straightforward to take advantage of thread-level parallelism in traversal, but SIMD operations are more challenging. The two basic approaches used in the literature are *packet tracing* [WSBW01, RSH05] where each SIMD lane is mapped to a separate ray-thread, and MBVH traversal [DHK08, WBB08, EG08] where lanes correspond to child bounding boxes in a tree of high branching factor. Both techniques have limitations. It is hard to reach a good SIMD utilization in packet tracing when rendering complex visual effects, while MBVH performs best with 4-wide vectors, and wider vectors tend to give marginal benefits.

Current GPUs, in turn, are based on the Single Instruction Multiple Thread (SIMT) execution model, which maps programs written in multi-threaded idiom onto SIMD-like, wide vector hardware. When execution diverges, the hardware executes the instructions of both execution paths sequentially, such that threads on each path stall while the other pah executes. Memory access latencies are hidden by massive multithreading.

The use of commodity GPUs for ray tracing was first investigated in the seminal work by Purcell [PBMH02] and rapidly became the state of the art for high-performance ray tracing, due to the large amount of parallelism available. However, GPUs are still not a perfect match for traversal due to its high degree of execution divergence. The number of traversal steps and intersection tests varies significantly between rays, hurting utilization. GPU renderers incorporate techniques such as replacement of terminated rays [AL09] to mitigate the issue.

Recently, several academic and commercial hardware architectures have been proposed to accelerate ray tracing [DNL*17]. These can be split into programmable and fixed-function systems. Programmable systems aim to fix the weaknesses of commodity CPUs and GPUs in ray traversal, typically, through some form of MIMD execution. Fixed-function systems aim for further efficiency by being hardwired to perform tree traversal steps and intersection tests. Fixed-function hardware is often $2-3$ orders of magnitude more efficient than a programmable system [HQW*10], at the cost of flexibility.

Recent work on ray tracing architectures has a special focus on the memory hierarchy. The reason for this is that the architectures are aimed for the rendering of increasingly complex and ambitious lighting effects, which generate secondary rays with highly incoherent memory access patterns from ray to ray, reducing cache hit rates. As a result, traversal becomes highly memory-intensive. For example, on GPUs, incoherent secondary rays can be $2-10\times$ slower to traverse than primary rays [YKL17], owing to cache misses.

This Chapter studies the application of MBVHs to fixed-function ray tracing accelerators. We first review the literature on accelerator architectures, and MBVH ray traversal. Finally, the contributions of this Thesis to the literature are summarized.

4.1 Traversal Architectures

This Section reviews previous work on ray tracing accelerators, with focus on fixed-function systems. The reader is directed to a recent, extensive survey paper [DNL*17] for more information.

4.1.1 Programmable Platforms

The main aim in proposed programmable ray tracers is to handle applications with high thread-level parallelism and execution divergence. One approach is to build a *many-core* processor. An early programmable architecture, RPU [WSS05], consisted of a multicore of narrow SIMD processors with a special instruction set. In addition to ray traversal with packet tracing, the same hardware is used for ray generation, shading and limited dynamic scene support.

Govindaraju et al. [GDS*08] proposed a many-core architecture which was evaluated with configurations of up to 128 cores, based on the earlier Razor software ray tracing system [SMD*06]. The architecture is divided into 8-core *tiles*, each with a shared L2 cache and rendering-specific acceleration hardware. The individual cores are SPARC Niagara processors to allow prototyping of a part of the system on a multicore SPARC workstation.

A family of MIMD architectures developed at University of Utah [SKKB09, KSS*13, KSS*15, SGK*17], beginning with the TraX processor [SKKB09], consists of many simple processing cores with separate instruction streams. Cores stall on cache miss, but the cost of stalls is mitigated by sharing expensive resources such as floating-point units between multiple cores, such that the individual cores are very lightweight.

The MRTP [KKK12] architecture is a SIMT system similar to conventional GPUs, but can be reconfigured to split into narrower SIMT cores with separate instruction streams in order to handle divergent workloads. For example, a 12-wide SIMT core may be split into two 6-wide SIMT cores. A test chip was later fabricated for a similar hardware architecture [KKOK13].

Raman et al. proposed a ray tracing architecture based on SIMD stream filtering processors, StreamRay [RGD09]. The ray tracing process is divided into stream filtering tasks, each of which task is handled by a SIMD processor which maps ray-threads from an incoming stream onto SIMD lanes on the fly. This approach gives a high SIMD utilization and avoids divergence issues.

Some programmable architectures have been later modified to use custom pipelines for traversal, so that the flexibility of the programmable system is used for other tasks such as ray generation or shading. The D-RPU [WBS06] reworks the earlier fully-programmable RPU to handle inner node traversal and intersection with fixed-function hardware. Likewise, the STRaTA architecture [SFD09], based on TraX [SKKB09], has an option to connect arithmetic units, normally used by programmable processors, into special-purpose traversal and intersection pipelines.

4.1.2 Fixed-Function Accelerators

The process of ray traversal can be broken into primitive intersection tests and inner node traversals, and a natural structure for a hardware accelerator is to include separate pipelines for each task. SaarCOR [SWS02] could be considered the prototype for modern fixed-function ray tracers: it is split into *ray tracing cores* with Binary Space Partition (BSP) tree traversal and triangle intersection pipelines, as well as a programmable processor for ray generation and shading. The main themes in later work include *load balancing*, evolution in the used data structures and algorithms, and optimizations to the memory system.

Load balancing becomes an issue since the ratio of traversal steps to primitive intersection tests is scene-dependent and varies between individual rays − with triangle-heavy scenes, the intersection units may become a bottleneck and starve the traversal pipelines, and vice versa for node-heavy scenes. SaarCOR includes equal numbers of traversal and intersection units, and attempts to load-balance by controlling the depth of subdivision in the acceleration tree.

The T&I Engine by Nah et al. [NPP*11] allocates more traversal units than intersection units, and designs intersection units with low silicon area and throughput. Moreover, they split the triangle intersection into two stages, where the first stage is a cheap test allowing early rejection of obviously non-intersecting rays, and the second stage finishes the full intersection test. Fewer second-stage than first-stage units are allocated. Later architectures trend toward increasing simplicity. The SGRT [LSL*13] architecture still reserves multiple traversal units per intersection unit but uses a unified intersection pipeline, possibly since (in the Thesis Author's experience) BVH traversal results in fewer early returns from the primitive intersection test, has a unified intersection pipeline, and reserves multiple traversal units per intersection unit. The RayCore [NKK*14] has a unified pipeline which reuses some arithmetic units for both traversal and intersection.

4.1.3 Memory Access Schemes

Recently, attention in ray tracing hardware research has shifted toward optimizing the memory hierarchy. Two aspects of the memory system are important. Firstly, bandwidth to off-chip memory is a limited resource, and can become a performance bottleneck. Traffic to such memory also consumes significant energy consumption, often being the largest single component of energy consumption [KSS*15]. A fundamental approach to reducing the stress on external memory is to place small memories, *caches*, on-chip, such

that accesses to a frequently visited *working set* of addresses are served from the cache. Simple caching is efficient in primary ray tracing, but with incoherent secondary rays, hit rates fall, and ray tracing tends to become memory-limited.

Secondly, memory access latencies are large and unpredictable – up to hundreds of clock cycles in the case of an external memory access. While a thread of execution is waiting on a data read, the computation hardware needs to be supplied with other work that does not depend on the same read. A basic computer architecture approach is multithreading where, when a thread needs to wait for a memory read, computation switches to another thread.

4.1.3.1 Latency Hiding

All proposed hardware ray tracers make liberal use of multithreading to hide memory latencies. There are several variations on how to handle cache misses. The T&I Engine by Nah et al. [NPP*11] introduces a Ray Accumulation Unit (RAU) that accumulates rays that have undergone a cache miss. Rays waiting on the same read address are grouped on the same RAU row, and fed to the compute pipeline together when the corresponding read data arrives. The *reorder buffer* by Lee et al. [LSH*15] improves on the RAU by omitting storage of read data in the buffer, simplifying the control logic and eliminating stalls due to full RAU rows.

A simpler approach is to allow threads to continue after a cache miss, discarding the invalid results of traversal and intersection computations, and schedule the thread to retry the computations. This is called *looping until the next chance* by Nah et al. [NKK*14], and a similar scheme was proposed earlier for general-purpose GPU use as *retry buffers* by Kwon et al. [KSP*13]. In programmable ray tracers, an interesting multithreading scheme is seen TraX [SKKB09] and related MIMD processors: each thread has a separate compute core which stalls on cache misses, but expensive resources like FPUs are shared between multiple cores, and remain well utilized.

4.1.3.2 Treelet Scheduling

An influential work by Aila and Karras [AK10] proposes a cache-aware technique of *treelet scheduling* to reduce traffic. In treelet scheduling, the traversal data structure is split into treelets small enough to fit in a L1 cache, and traversal operates over many rays which are stored off-chip. Each traversal core focuses on one treelet at a time, streaming in the rays queuing for that treelet, and then streaming the rays off-chip to queues of other treelets as they exit the active treelet. The approach appears inspired by earlier software out-of-core ray tracing schemes, e.g., Pharr et al. [PKGH97] subdivide the scene into a voxels with separate acceleration trees, allocate ray queues for each voxel, and page in the contents of each voxel from the disk as needed. Kopta et al. [KSS*15] propose a variant of treelet scheduling that uses a smaller number of rays which are stored on-chip, eliminating ray traffic, but reducing the amount of coherency that can be extracted from the active ray group. Recently, Shurko et al. [SGK*17] develop another variant called *dual streaming*, where traversal is guaranteed to visit each treelet at most once in deterministic order.

4.1.3.3 Tree Compression

Another approach to traffic reduction is to compress the acceleration data structure, allowing more node data to fit in caches and improving hit rates. Recently, compressed data structures have been proposed that can be traversed with cheap reduced-precision

arithmetic [Kee14, VAMS16]. Compression is a broad subject, and is discussed in detail in Chapter 3.

4.1.3.4 Stack Optimizations

Given a high degree of multithreading, per-thread traversal stacks may take up too much memory to store on-chip, and stack traffic can account for a large fraction of the total. Several works propose software methods for stackless [HDW*11, ÁSK14] or short-stack [Lai10] traversal to eliminate full stacks. So far these methods have given inferior performance compared to stack-based traversal since they incur extra node visits and other computational overhead, but are often motivated by potential use in hardware accelerators. The hardware compressed traversal scheme by Vaidyanathan et al. [VAMS16] uses a short-stack to compensate for the large stack elements required by the traversal algorithm. Aila and Karras [AK10] propose a hardware *stack cache* which eliminates most external stack accesses with a small silicon footprint.

4.1.3.5 Streaming Hybrid Ray Tracing

Primary ray tracing with a simple pinhole camera model is equivalent to rasterization. Since it is doubtful that even hardware primary ray tracing can beat the efficiency of a conventional GPU in rasterization, there is interest in *hybrid ray tracing*, where primary ray tracing is replaced with rasterization, and only secondary rays are actually traversed. Lee et al. [LHS*15] propose an synthesis of a conventional mobile GPU and a hardware ray tracer optimized for hybrid ray tracing. The architecture is built around the mobile GPU technique of tile-based rendering [AMS08]. Tiles in the primary ray buffer which will have associated secondary rays, are streamed on-chip to the hardware ray tracer, reducing off-chip communication.

4.2 Multi-Bounding Volume Hierarchies

As mentioned in the previous Section, development of traversal algorithms has been guided by trends in processing hardware. For example, as commodity CPUs began to incorporate SIMD vector instruction sets such as Streaming SIMD Extensions (SSE) and later Advanced Vector Extensions (AVX), a large amount of research went to exploiting these instructions in ray traversal. The first mainstream technique for SIMD traversal was *packet tracing* [WSBW01], in which the traversal loop operates on ray packets instead of single rays. Each ray in a packet is mapped to a SIMD lane, and vector instructions are used to intersection test all rays of the packet against the same node or primitive in parallel.

Packet tracing works well in workloads like primary rays which exhibit high coherency, i.e., rays in the same packet are likely to intersect the same nodes and primitives. However, efficient packet tracing becomes difficult to achieve in incoherent workloads like path tracing, where rays tend to diverge, dropping SIMD utilization. A wide variety of workarounds has been developed to extract *hidden coherency* in the rendering workload by, e.g., traversing larger ray groups and periodically sorting them to find coherent packets [MMAM07], or organizing rays in a separate data structure to search for coherence [AK87]. Similarly to treelet scheduling, one approach was to divide the scene into subvolumes, limiting traversal to rays that are located in the same volume [PKGH97].

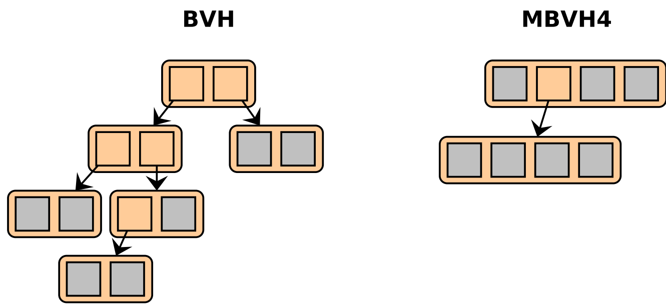


Figure 4.1: 4-way MBVH tree contrasted with a binary BVH tree, both organizing 7 leaf nodes. In this case, MBVH saves memory: BVH needs 6 nodes á 64B for a memory consumption of 384B, while MBVH uses 2 nodes á 128B for a total of 256B.

Given the difficulty of packet tracing secondary rays, several research groups arrived independently at a new SIMD parallelization scheme ca. 2007, which leverages BVH tree with a high branching factor, e.g., 4 or 8 children per node. Vector instructions are then used in traversal to test a single ray against all child AABBs of a node in parallel. This approach was introduced by Dammertz et al. [DHK08] as Quad Bounding Volume Hierarchy (QBVH), and Wald et al. [WBB08] as well as Ernst and Greiner [EG08] as MBVH, and quickly became a standard technique in CPU ray tracing. The development of MBVH was probably prompted by the adoption of BVH: most of the literature on packet tracing was built on kd-trees where a similar scheme is not as straightforward.

Compared to a binary BVH storing two child AABBs per node, a four-way MBVH requires twice as many intersection tests to traverse a node, but traversing the ray typically requires ca. 2× less node traversals, as the hierarchy is shallower. In other words, MBVH traversal takes roughly as much computation as BVH, but grouped in larger units that are suitable for vector arithmetic. Moreover, MBVH can somewhat reduce memory consumption and improve cache hit rates [DHK08]. As shown in Fig. 4.1, though, e.g., a 4-way MBVH node uses 2× the memory of a binary BVH node, the number of nodes can be reduced to less than half. MBVH implementations often also build trees with large leafs in order to SIMD acceleration primitive intersection tests, further reducing memory footprint [WBB08].

MBVHs is the workhorse of state-of-the-art CPU ray tracers [WWB*14]. Recent research also shows benefits in GPU ray tracing [Gut14], even though vector instructions are unavailable on GPU – partly due to reduced memory traffic, and partly because MBVH node traversal is a better fit for out-of-order execution schemes on modern GPUs. 4-way branching is a sweet spot in terms of efficiency: wider MBVHs start to do significantly more computation per ray than binary trees, though they can still give performance benefits by using available wide vector units [Áfr13]. However, it may be more efficient to combine 4-wide MBVH with packet tracing [WWB*14], or *ray stream tracing* where coherence is extracted from larger ray groups [BAM14].

4.2.1 Construction

Approaches to building MBVHs can be divided into *collapsing* a binary BVH into a MBVH, and *native builds* which directly output a MBVH. A basic way to collapse BVHs is to remove alternating tree layers to form a 4-way MBVH [EG08, DHK08], or $\frac{2}{3}$ of layers for an 8-way tree [GL10]. 'Afra et al. [Áfr13] collapse nodes by repeatedly removing the child with the highest surface, and pulling the corresponding grandchildren into the node. Wald et al. [WBB08] propose a SAH-guided collapsing algorithm which merges child nodes into the parent according to their SAH heuristic cost, until the parent is full. This is combined with other optimization steps to avoid being stuck in local optima.

Pinto et al. [Pin10] find a sequence of collapsing operations that globally optimizes a cost heuristic for a MBVH. Ylitie et al. [YKL17] propose a slow, high-quality bottom-up collapsing algorithm which computes, bottom-up, the optimal SAH cost reachable for each subtree through collapsing operations, and then generates top-down the splits that realize the optimal cost. The algorithm takes minutes to run on large scenes, but gives superior tree quality compared to previous fast methods.

Wald et al. [WBB08] give a recursive, top-down build algorithm which is reported to give similar tree quality as collapsing. When processing a node, the algorithm finds the best axis-aligned split for each child, and applies the split that gives the best SAH improvement, until the node is full. The algorithm can be regarded as a generalization of the binned SAH sweep [Wal07] for higher branching factors.

4.2.2 Traversal

The main subtasks of BVH inner node traversal are *intersection tests* against a node's child AABBs and *stack operations* to push the intersecting children onto the traversal stack. An important optimization is to push the children so that they are traversed in *front-to-back order*: the ray then often hits occluding geometry in the closer child, and avoids visiting the farther child entirely. Intersection tests are trivial to vectorize, but stack operations and front-to-back ordering are more challenging to express as SIMD operations.

4.2.2.1 Front-to-Back Sorting

Front-to-back traversal can be performed by computing the parametric distance to each child AABB being intersection tested, and sorting the hit children based on this distance. As fast distance sorting is nontrivial, Wald et al. [WBB08] opt to store an unordered stack, and scan the stack for the closest element on each pop. Dammertz et al. [DHK08] perform a SIMD vector sort [FAN07]. Guthe et al. [Gut14] sort children with a *sorting network* to improve instruction-level parallelism in their GPU ray tracer. Áfra et al. [Áfr13] selects the sorting algorithm adaptively based on the number of hit children: $1-2$ hits are handled as special cases, $3-4$ with a sorting network, and 5 or more with an insertion sort.

4.2.2.2 Sign Heuristic

In addition to sorting children according to their hit point distance from ray origin (i.e., the *distance heuristic*), similar results can be obtained with the *sign heuristic* [Mah05]: if a BVH is built based on top-down splits, the split axis is stored. If the ray direction is positive along the split axis, the children are visited in the order of storage; if it is

negative, the order is flipped. As an alternative to sorting, Dammertz et al. [DHK08] experiment with building their MBVHs via collapsing and storing the topology and split axes of the original binary BVH treelet for each node; this stored metadata is then used for sign heuristic traversal. A similar approach was used by Ernst and Greiner [EG08]. Recently, Fuetterling et al. [FLP*17] use a similar approach implemented entirely with vector instructions and for a wide branching factor.

4.2.2.3 Approximate Orders

Some works opt to forego exact sorting. Wald et al. [WBB08] omit distance sorting as overly expensive: they only find the closest hit child and push the other hit children to the stack unordered. It should be noted that this only differs from a full sort if more than 2 children are hit, which is a rare case [Áfr13]. Áfra et al. [ÁSK14], likewise, only guarantee that the closest child is traversed first; subsequent sibling nodes are traversed in fixed order by following *skip pointers* between siblings. Ylitie et al. [YKL17], inspired by Garanzha and Loop [GL10], precompute the octant of each child node relative to the parent box, and traverse the octants in distance order.

4.2.2.4 Occlusion Rays

When tracing *shadow* and *ambient occlusion* [Bun05] rays, it is acceptable to find any intersection instead of the closest one. Guthe et al. [Gut14] note that, in this case, front-to-back sorting can be omitted. Ogaki et al. [ODJ16] pre-sort MBVH children in a *surface area traversal order* which further improves performance.

4.3 Thesis Contribution

To summarize the above review, ray-tracing accelerator architectures have been proposed that are estimated give ray traversal performances similar to desktop GPUs in a mobile environment [LSL*13, NKK*14], or significantly higher if given a desktop-level power budget [Kee14]. Most architectures rely on fixed-function hardware for tree traversal. Shallow BVHs are in mainstream use in CPU ray tracing [WWB*14], and have recently been found beneficial on GPUs [Gut14, YKL17] but, so far, all proposed ray tracing accelerators use binary trees.

In this Thesis, MBVHs are applied to hardware-accelerated BVH traversal. Specifically, in [P4], MBVH is evaluated for a simulated generic fixed-function hardware accelerator similar to SaarCOR [SWS02], SGRT [LSL*13] or D-RPU [WBS06]. The chosen multi-threading scheme is the looping-until-the-next-chance used in the RayCore [NKK*14]. The simulation includes timing, silicon area and energy models.

The main finding of [P4] is that MBVHs are a clear low-hanging fruit, largely due to reduced memory traffic. The result holds both for energy efficiency and performance per area. Another contribution of [P4] is that MBVH distance sorting − a target of sophisticated optimizations in software ray tracers [Áfr13, FLP*17] − is straightforward to implement in hardware while keeping a fully pipelined throughput, by means of, e.g., a cheap hardware sorting network.

[P4] leaves as an open question whether the benefits from MBVHs are cumulative with optimizations like tree compression, which is discussed in more detail in Chapter 3, and treelet scheduling. Recently, Ylitie et al. [YKL17] were able to significantly

accelerate software GPU ray tracing with Compressed Multi Bounding Volume Hierarchies (CMBVHs), suggesting that this is the case at least for compression.

5 Conclusion

Real-time ray tracing is a long-standing research goal in computer graphics, motivated by, e.g., global illumination effects, video games and photorealistic AR. Perhaps the clearest potential motivation is that ray tracing could render some common effects such as dynamic shadows and reflections at a lower energy cost than multi-pass rasterization methods [KKW*13], but such savings are likely to require specialized hardware architectures. A wide variety of such architectures has been proposed [DNL*17].

Recent research on ray tracing GPU architectures has been centered on reducing off-chip memory traffic, which can account for a large fraction of operating energy, and often becomes a performance bottleneck. Two promising classes of optimizations have emerged which promise to reduce memory traffic by an order of magnitude compared to a naive ray tracer. These are the use of treelet-based memory access schedules [AK10, KSS*15, SGK*17], and the compression of data structures used in ray tracing [Kee14, VAMS16, LV16]. This mirrors the development of mobile GPUs which incorporate a slew of memory traffic optimizations such as tile-based rendering [AMS08].

Coping with animated scenes is a special challenge in ray tracing, where the data structures used for rendering need to be updated for each animation frame, usually by rebuilding them from scratch. There is a rich literature on fast build algorithms for this purpose. The results of this Thesis [P2] underline that the cost of tree build is significant, making it an interesting target for hardware acceleration. On mobile platforms, custom hardware seems to be a prerequisite for handling nontrivial animated scenes in real time. Moreover, tree build is even more memory-intensive than rendering, placing the emphasis of architecture design on memory traffic optimizations.

The rendering optimizations mentioned above have a drawback of causing additional complications to tree builds and updates. For instance, treelet scheduling [AK10] requires trees to be divided into suitable treelets, while compressed trees are clearly more challenging to update than their basic variants. Moreover, these optimizations make the cost of tree updates more pronounced on the system level by way of Amdahl's law, by significantly shrinking the cost of rendering, while the cost of update remains largely intact.

Energy-efficiency and memory bandwidth are prime considerations in mobile systems, which are constrained to tiny power budgets and memory bandwidths compared to the desktop, by virtue of being hand held and battery-powered. It is interesting to note that, due to trends in CMOS technology [BC11], energy and bandwidth optimizations are also becoming increasingly relevant on the desktop. Over the past decade, the arithmetic performance of GPUs has grown much faster than bandwidth, and previously compute-limited algorithms are becoming memory-limited. As a case in the point, a recent state-of-the-art software ray tracer [YKL17] incorporates tree compression, which

was previously advanced mainly as a hardware optimization. Similarly, bandwidth optimizations for tree construction have the best effect in mobile, hardware construction, but may also serve to inform the design of future software and desktop builders.

5.1 Main Results

This Thesis explored techniques for hardware acceleration of ray tracing, with focus on real-time rendering of large-scale animated scenes. This Thesis concentrated on mobile accelerators operating in a limited power envelope, but the techniques are also applicable to other environments.

A new hardware tree builder architecture named MergeTree was proposed in publications [P1,P2]. The builder adapts the fast GPU tree construction algorithm LBVH [HHS06] into a lightweight hardware pipeline. The hardware version of the algorithm is optimized for memory bandwidth economy, and consumes 3× less bandwidth than a software implementation on GPU. The main novel components enabling the memory savings are a memory-frugal primitive sorting subsystem, and a streaming method for hierarchy emission and AABB computation. LBVH produces low-quality trees, but measurements in publication [P2] show that it can reduce system energy compared to a state of the art binned SAH builder [DFM13], given a fairly large animated scene and a limited amount of secondary rays. This is because the energy savings in tree construction in large scenes are enough to offset the increased cost of traversal.

Publication [P3] investigated efficient updates of BVH trees compressed with *incremental encoding* [Kee14, VAMS16, LV16], a state-of-the-art approach to bandwidth reduction in ray tracing. The main result is that if the output of a tree builder can be compressed in a streaming manner before writing it off-chip, significant traffic savings are available. Given that memory traffic accounts for over 90% of the energy of fast tree updates [P2], compressed trees are then cheaper to update than uncompressed ones. Streaming compression is straightforward in top-down updates, but the bottom-up output order used in the fastest hardware updaters [KIS*12][P2] poses technical difficulties, as the compressed encoding of a node depends on its chain of parents. An algorithm is proposed in [P3] which speculatively encodes each node based on estimates of parameters that would be derived from the parent nodes. In case the estimates are invalid for some node, the algorithm later backtracks to repair it.

The fixed-function builders proposed in publications [P1,P2,P3] give the largest benefits for mobile systems, where software builds are only feasible for trivial scenes. Moreover, they may be used in desktop ray tracing to free up limited computational resources for ray traversal and shading. Often, efficiency gains from the use of fixed-function pipelines are offset by limited flexibility. However, flexibility is enhanced in this case by processing AABBs instead of raw primitive data. As with Doyle et al. [DFM13], this allows the proposed pipelines to handle any type of primitive for which an AABB can be computed, for example, volume data [DTM17].

Finally, [P4] applied MBVHs, a workhorse method of CPU ray tracing, to hardware accelerators. This is found to be a low-hanging fruit that is straightforward to implement and improves energy and area efficiency. Evaluation is done on an energy, area and timing model of a generic fixed-function accelerator architecture similar to SaarCOR [SWS02], the T&I Engine [NPP*11] and SGRT [LSL*13]. The model developed for this work was later used for system-level evaluation of the aforementioned MergeTree builder [P2].

5.2 Open Research Issues

The main results of this Thesis came from representation of well-known algorithms in terms of streaming processes, which allow memory traffic savings compared to the GPU idiom of performing tasks in multiple passes that read and write large intermediate data buffers. It is interesting whether stream representation could stretch further to compute, e.g., more sophisticated sorting-based build algorithms such as ATRBVH [DP15], spatial split insertion, or hierarchy collapsing into an MBVH. Moreover, the fixed-function realizations of streaming algorithms are inflexible. It is interesting whether the algorithms could be set up to run on a programmable processor − even an exotic one such as a dataflow machine − and still reach a reasonable performance.

From a higher-level point of view, despite the volume of work on the subject, so far the evidence is not very strong that the overall idea of ray tracing as an energy optimization is feasible − i.e., that it can give improved visual effects in a given energy budget compared to rasterization-based methods, or in the mobile case, that it can render workloads approximating practical real-time applications within a mobile power and bandwidth budget. One difficulty in rigorous comparison is that conventional GPUs are the products of cumulative decades of work by large industrial design teams, and it would be difficult to match this level of workmanship in an academic prototype GPU. Moreover, real-time graphics techniques for conventional GPUs are well understood, while the designers of nontrivial example applications of real-time ray tracing would be starting partly from scratch. The level of confidence in the area could be increased by introducing more energy analysis of hardware designs and adding comparisons to the available power budget, especially for mobile designs. This Thesis made some efforts in this direction by, e.g., performing energy analysis and comparing results to a mobile power envelope in [P2].

A promising avenue for future work is to accelerate computations beside simple rendering with ray tracing hardware. For example, photorealistic AR, where virtual objects blend in seamlessly with the real environment, requires estimates of the geometry, lighting and materials in the scene. Several state-of-the-art methods for producing these estimates [RTKPS16, KPR*15] make heavy use of ray traversal, and are computationally expensive. Consequently, hardware ray tracers could be used to improve the performance of AR, which has a wide variety of potential applications [vKP10]. The tree constructors proposed in this Thesis [P1,P2,P3], in turn, would be interesting to apply to point set processing and collision detection. It is straightforward to modify an LBVH builder to output k-d trees of points [Kar12], which are widely used in point set processing. Collision detection, meanwhile, tends to directly use BVHs. Hardware units have been proposed for both tasks [RHAZ06, HGBG08], identifying hardware-accelerated tree construction as future work.

Bibliography

[Áfr12] Áfra A. T.: Interactive ray tracing of large models using voxel hierarchies. In *Computer Graphics Forum* (2012), vol. 31, pp. 75–88.

[Áfr13] Áfra A. T.: *Faster Incoherent Ray Traversal Using 8-Wide AVX Instructions.* Tech. rep., Babes-Bolyai University, 2013.

[AK87] Arvo J., Kirk D.: Fast ray tracing by ray classification. In *ACM SIGGRAPH Computer Graphics* (1987), vol. 21, pp. 55–64.

[AK10] Aila T., Karras T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics* (2010), pp. 113–122.

[AKL13] Aila T., Karras T., Laine S.: On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics* (2013), pp. 101–107.

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics* (2009), pp. 145–149.

[AMS08] Akenine-Möller T., Strom J.: Graphics processing units for handhelds. *Proceedings of the IEEE 96*, 5 (2008), 779–789.

[Ape14] Apetrei C.: Fast and Simple Agglomerative LBVH Construction. In *Proc. Computer Graphics and Visual Computing Conf.* (2014).

[ÁSK14] Áfra A. T., Szirmay-Kalos L.: Stackless multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140.

[ÁWBW16] Áfra A. T., Wald I., Benthin C., Woop S.: Embree ray tracing kernels: overview and new features. In *SIGGRAPH Talks* (2016), p. 52.

[BAM14] Barringer R., Akenine-Möller T.: Dynamic ray stream traversal. *ACM Transactions on Graphics 33*, 4 (2014), 151.

[BC11] Borkar S., Chien A. A.: The future of microprocessors. *Communications of the ACM 54*, 5 (2011), 67–77.

[BEM10] Bauszat P., Eisemann M., Magnor M. A.: The minimal bounding volume hierarchy. In *Proc. Int. Symp. Vision, Modeling and Visualization* (2010), pp. 227–234.

[BHH13] Bittner J., Hapala M., Havran V.: Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100.

[Bik07] Bikker J.: Real-time ray tracing through the eyes of a game developer. In *Proc. IEEE Symp. Interactive Ray Tracing* (2007), pp. 1–10.

[Bun05] Bunnell M.: Dynamic ambient occlusion and indirect lighting. In *GPU Gems*, vol. 2. NVidia, 2005, pp. 223–233.

[BWN*15] Benthin C., Woop S., Niessner M., Selgrad K., Wald I.: Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proc. High-Performance Graphics* (2015), pp. 5–12.

[BWSF06] Benthin C., Wald I., Scherbaum M., Friedrich H.: Ray tracing on the Cell processor. In *Proc. IEEE Symp. Interactive Ray Tracing* (2006), pp. 15–23.

[BWW*12] Benthin C., Wald I., Woop S., Ernst M., Mark W. R.: Combining single and packet-ray tracing for arbitrary ray distributions on the intel MIC architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (2012), 1438–1448.

[Cat74] Catmull E.: *A subdivision algorithm for computer display of curved surfaces*. Tech. rep., Utah University, Salt Lake City, USA, 1974.

[CO14] Casper J., Olukotun K.: Hardware acceleration of database operations. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays* (2014), pp. 151–160.

[COCSD03] Cohen-Or D., Chrysanthou Y. L., Silva C. T., Durand F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 412–431.

[CPC84] Cook R. L., Porter T., Carpenter L.: Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics* (1984), vol. 18, pp. 137–145.

[CSE06] Cline D., Steele K., Egbert P.: Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools 11*, 4 (2006), 61–71.

[DFM13] Doyle M., Fowler C., Manzke M.: A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics 32*, 4 (2013), 139:1–10. (Proc. SIGGRAPH 2013).

[DHK08] Dammertz H., Hanika J., Keller A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1233.

[DKT98] DeRose T., Kass M., Truong T.: Subdivision surfaces in character animation. In *Proc. SIGGRAPH* (1998), pp. 85–94.

[DKY16] Du P., Kim Y. J., Yoon S.-E.: TSS BVHs: Tetrahedron swept sphere BVHs for ray tracing subdivision surfaces. In *Computer Graphics Forum* (2016), vol. 35, pp. 279–288.

[DNL*17] Deng Y., Ni Y., Li Z., Mu S., Zhang W.: Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys 50*, 4 (2017), 58.

[DP15] Domingues L. R., Pedrini H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics* (2015), pp. 13–20.

[DTM17] Doyle M. J., Tuohy C., Manzke M.: Evaluation of a BVH construction accelerator architecture for high-quality visualization. *IEEE Transactions on Multi-Scale Computing Systems* (2017). Early access.

[EBGM12] Eisemann M., Bauszat P., Guthe S., Magnor M.: Geometry presorting for implicit object space partitioning. *Computer Graphics Forum 31*, 4 (2012), 1445–1454.

[EG08] Ernst M., Greiner G.: Multi bounding volume hierarchies. In *Proc. IEEE Symp. Interactive Ray Tracing* (2008), pp. 35–40.

[ESV96] Evans F., Skiena S., Varshney A.: Optimizing triangle strips for fast rendering. In *Proc. Conf. Visualization* (1996), pp. 319–326.

[EW11] Ernst M., Woop S.: Ray tracing with shared-plane bounding volume hierarchies. *Journal of Graphics, GPU, and Game Tools 15*, 3 (2011), 141–151.

[FAN07] Furtak T., Amaral J. N., Niewiadomski R.: Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proc. ACM Symp. Parallel Algorithms and Architectures* (2007), pp. 348–357.

[FD09] Fabianowski B., Dingliana J.: Compact BVH storage for ray tracing and photon mapping. In *Proc. Eurographics Ireland Workshop* (2009), pp. 1–8.

[FGD*06] Friedrich H., Günther J., Dietrich A., Scherbaum M., Seidel H.-P., Slusallek P.: Exploring the use of ray tracing for future games. In *Proc. ACM SIGGRAPH Symp. Videogames* (2006), pp. 41–50.

[FLP*17] Fuetterling V., Lojewski C., Pfreundt F.-J., Hamann B., Ebert A.: Accelerated single ray tracing for wide vector units. In *Proc. High-Performance Graphics* (2017), p. 6.

[GA05] Galin E., Akkouche S.: Fast processing of triangle meshes using triangle fans. In *Proc. Int. Conf. Shape Modeling and Applications* (2005), pp. 326–331.

[GBDAM15] Ganestam P., Barringer R., Doggett M., Akenine-Möller T.: Bonsai: Rapid bounding volume hierarchy generation using mini trees. *Journal of Computer Graphics Techniques 4*, 3 (2015).

[GD16] Ganestam P., Doggett M.: SAH guided spatial split partitioning for fast BVH construction. *Computer Graphics Forum 35*, 2 (2016), 285–293.

[GDS*08] Govindaraju V., Djeu P., Sankaralingam K., Vernon M., Mark W. R.: Toward a multicore architecture for real-time ray-tracing. In *Proc. IEEE/ACM Int. Symp. Microarchitecture* (2008), pp. 176–187.

[GHFB13] Gu Y., He Y., Fatahalian K., Blelloch G.: Efficient BVH construction via approximate agglomerative clustering. In *Proc. High-Performance Graphics* (2013), pp. 81–88.

[GL10] Garanzha K., Loop C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. In *Computer Graphics Forum* (2010), vol. 29, pp. 289–298.

[GPBG11] Garanzha K., Premože S., Bely A., Galaktionov V.: Grid-based SAH BVH construction on a GPU. *The Visual Computer 27*, 6-8 (2011), 697–706.

[GPM11] Garanzha K., Pantaleoni J., McAllister D.: Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics* (2011), pp. 59–64.

[GS87] Goldsmith J., Salmon J.: Automatic creation of object hierarchies for ray tracing. *Computer Graphics and Applications 7*, 5 (1987), 14–20.

[Gut14] Guthe M.: Latency considerations of depth-first GPU ray tracing. In *Proc. Eurographics (Short Papers)* (2014), pp. 53–56.

[HDW*11] Hapala M., Davidovič T., Wald I., Havran V., Slusallek P.: Efficient stack-less BVH traversal for ray tracing. In *Proc. Spring Conf. Computer Graphics* (2011), pp. 7–12.

[HGBG08] Heinzle S., Guennebaud G., Botsch M., Gross M.: A hardware processing unit for point sets. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware* (2008), pp. 21–31.

[HH10] Havel J., Herout A.: Yet faster ray-triangle intersection (using SSE4). *IEEE Transactions on Visualization and Computer Graphics 16*, 3 (2010), 434–438.

[HHS06] Havran V., Herzog R., Seidel H.-P.: On the fast construction of spatial hierarchies for ray tracing. In *Proc. IEEE Symp. Interactive Ray Tracing* (2006), pp. 71–80.

[HK07] Hanika J., Keller A.: Towards hardware ray tracing using fixed point arithmetic. In *Proc. IEEE Symp. Interactive Ray Tracing* (2007), pp. 119–128.

[HLS*15] Hwang S. J., Lee J., Shin Y., Lee W.-J., Ryu S.: A mobile ray tracing engine with hybrid number representations. In *Proc. SIGGRAPH Asia Symp. Mobile Graphics and Interactive Applications* (2015), p. 3.

[HMB17] Hendrich J., Meister D., Bittner J.: Parallel BVH construction using progressive hierarchical refinement. *Computer Graphics Forum 36*, 2 (2017), 487–494.

[HMF07] Hunt W., Mark W. R., Fussell D.: Fast and lazy build of acceleration structures from scene hierarchies. In *Proc. IEEE Symp. Interactive Ray Tracing* (2007), pp. 47–54.

[HQW*10] Hameed R., Qadeer W., Wachs M., Azizi O., Solomatnikov A., Lee B. C., Richardson S., Kozyrakis C., Horowitz M.: Understanding sources of inefficiency in general-purpose chips. *ACM SIGARCH Computer Architecture News 38*, 3 (2010), 37–47.

[HRB*09] Heinly J., Recker S., Bensema K., Porch J., Gribble C.: Integer ray tracing. *Journal of Graphics, GPU, and Game Tools 14*, 4 (2009), 31–56.

[HSO07] Harris M., Sengupta S., Owens J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems*, vol. 3. NVidia, 2007, pp. 851–876.

[HSPE06] Hartstein A., Srinivasan V., Puzak T. R., Emma P. G.: Cache miss behavior: is it $\sqrt{2}$? In *Proc. Conf. Computing Frontiers* (2006), pp. 313–320.

[Huf52] Huffman D. A.: A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[IK07] Ioannou A., Katevenis M. G.: Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking 15*, 2 (2007), 450–461.

[IWP07] Ize T., Wald I., Parker S. G.: Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proc. Eurographics Conf. Parallel Graphics and Visualization* (2007), pp. 101–108.

[KA13] Karras T., Aila T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. High-Performance Graphics* (2013), pp. 89–99.

[Kaj86] Kajiya J. T.: The rendering equation. In *ACM SIGGRAPH Computer Graphics* (1986), vol. 20, pp. 143–150.

[Kar12] Karras T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. High-Performance Graphics* (2012), pp. 33–37.

[KBK*10] Kim T.-J., Byun Y., Kim Y., Moon B., Lee S., Yoon S.-E.: HCCMeshes: Hierarchical-culling oriented compact meshes. *Computer Graphics Forum 29*, 2 (2010), 299–308.

[Kee14] Keely S.: Reduced precision hardware for ray tracing. In *Proc. High-Performance Graphics* (2014), pp. 29–40.

[KFF*15] Keller A., Fascione L., Fajardo M., Georgiev I., Christensen P. H., Hanika J., Eisenacher C., Nichols G.: The path tracing revolution in the movie industry. In *SIGGRAPH Courses* (2015), pp. 24–1.

[KIS*12] Kopta D., Ize T., Spjut J., Brunvand E., Davis A., Kensler A.: Fast, effective BVH updates for animated scenes. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games* (2012), pp. 197–204.

[KKK12] Kim H.-Y., Kim Y.-J., Kim L.-S.: MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *IEEE Journal of Solid-State Circuits 47*, 2 (2012), 518–535.

[KKOK13] Kim H.-Y., Kim Y.-J., Oh J.-H., Kim L.-S.: A reconfigurable SIMT processor for mobile ray tracing with contention reduction in shared memory. *IEEE Transactions on Circuits and Systems I: Regular Papers 60*, 4 (2013), 938–950.

[KKW*13] Keller A., Karras T., Wald I., Aila T., Laine S., Bikker J., Gribble C., Lee W.-J., McCombe J.: Ray tracing is the future and ever will be... In *SIGGRAPH Courses* (2013), p. 9.

[KM90] Katajainen J., Mäkinen E.: Tree compression and optimization with applications. *International Journal of Foundations of Computer Science 1*, 04 (1990), 425–447.

[KMKY10] Kim T.-J., Moon B., Kim D., Yoon S.-E.: RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2010), 273–286.

[Knu99] Knuth D. E.: *The Art of Computer Programming: Volume 3: Sorting and Searching*, vol. 3. 1999.

[KPR*15] Kähler O., Prisacariu V. A., Ren C. Y., Sun X., Torr P., Murray D.: Very high frame rate volumetric integration of depth images on mobile devices. *IEEE Transactions on Visualization and Computer Graphics 21*, 11 (2015), 1241–1250.

[KSP*13] Kwon K., Son S., Park J., Park J., Woo S., Jung S., Ryu S.: Mobile GPU shader processor based on non-blocking coarse grained reconfigurable arrays architecture. In *Proc. Int. Conf. Field-Programmable Technology* (2013), pp. 198–205.

[KSS*13] Kopta D., Shkurko K., Spjut J., Brunvand E., Davis A.: An energy and bandwidth efficient ray tracing architecture. In *Proc. High-Performance Graphics* (2013), pp. 121–128.

[KSS*15] Kopta D., Shkurko K., Spjut J., Brunvand E., Davis A.: Memory considerations for low energy ray tracing. *Computer Graphics Forum 34*, 1 (2015), 47–59.

[KSY14] Kim T.-J., Sun X., Yoon S.-E.: T-rex: Interactive global illumination of massive models on heterogeneous computing resources. *IEEE Transactions on Visualization and Computer Graphics 20*, 3 (2014), 481–494.

[KT11] Koch D., Torresen J.: FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays* (2011), pp. 45–54.

[KTO11] Kontkanen J., Tabellion E., Overbeck R. S.: Coherent out-of-core point-based global illumination. *Computer Graphics Forum 30*, 4 (2011), 1353–1360.

[KVJT16] Koskela M., Viitanen T., Jääskeläinen P., Takala J.: Half-precision floating-point ray traversal. In *Proc. Joint Conf. Computer Vision, Imaging and Computer Graphics Theory and Applications (1: GRAPP)* (2016), pp. 171–178.

[Lai10] Laine S.: Restart trail for stackless BVH traversal. In *Proc. High-Performance Graphics* (2010), pp. 107–111.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *ACM SIGGRAPH Computer Graphics* (1987), vol. 21, pp. 163–169.

[LDG17] Li Z., Deng Y., Gu M.: Path compression kd-trees with multi-layer parallel construction a case study on ray tracing. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games* (2017), p. 16.

[LDNL15] Liu X., Deng Y., Ni Y., Lil Z.: FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *Proc. Conf. Design, Automation and Test in Europe* (2015), pp. 1595–1598.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LHS*15] Lee W.-J., Hwang S. J., Shin Y., Yoo J.-J., Ryu S.: An efficient hybrid ray tracing and rasterizer architecture for mobile gpu. In *Proc. SIGGRAPH Asia Symp. Mobile Graphics and Interactive Applications* (2015), p. 2.

[LKA13] Laine S., Karras T., Aila T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proc. High-Performance Graphics* (2013), pp. 137–143.

[LL13] Lanman D., Luebke D.: Near-eye light field displays. *ACM Transactions on Graphics 32*, 6 (2013), 220.

[LMK17] Lin M. C., Manocha D., Kim Y. J.: Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*. CRC Press, 2017. To appear.

[LSH*15] Lee W.-J., Shin Y., Hwang S. J., Kang S., Yoo J.-J., Ryu S.: Reorder buffer: An energy-efficient multithreading architecture for hardware MIMD ray traversal. In *Proc. High-Performance Graphics* (2015), pp. 21–32.

[LSL*13] Lee W., Shin Y., Lee J., Kim J., Nah J.-H., Jung S., Lee S., Park H., Han T.: SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. High-Performance Graphics* (2013), pp. 109–119.

[Lue03] Luebke D. P.: *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[LV16] Liktor G., Vaidyanathan K.: Bandwidth-efficient BVH layout for incremental hardware traversal. In *Proc. High-Performance Graphics* (2016), pp. 51–61.

[LYM07] Lauterbach C., Yoon S.-E., Manocha D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. IEEE Symp. Interactive Ray Tracing* (2007), pp. 19–26.

[LYMT06] Lauterbach C., Yoon S.-E., Manocha D., Tuft D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proc. IEEE Symp. Interactive Ray Tracing* (2006), pp. 39–46.

[LYTM08] Lauterbach C., Yoon S.-e., Tang M., Manocha D.: ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum 27*, 4 (2008), 1313–1321.

[Ma05] Ma W.: Subdivision surfaces for CAD—an overview. *Computer-Aided Design 37*, 7 (2005), 693–709.

[Mah05] Mahovsky J. A.: *Ray tracing with reduced-precision bounding volume hierarchies.* PhD thesis, University of Calgary, 2005.

[MB90] MacDonald J. D., Booth K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166.

[MB16] Meister D., Bittner J.: Parallel BVH construction using k-means clustering. *The Visual Computer 32*, 6-8 (2016), 977–987.

[MB17] Meister D., Bittner J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics* (2017). Early access.

[MDPN16] McCombe J. A., Dwyer A., Peterson L. T., Nesse N.: Systems and methods for 3-d scene acceleration structure creation and updating, Aug. 30 2016. US Patent 9,430,864.

[MMAM07] Mansson E., Munkberg J., Akenine-Moller T.: Deep coherent ray tracing. In *Proc. IEEE Symp. Interactive Ray Tracing* (2007), pp. 79–85.

[MVCK17] Mashimo S., Van Chu T., Kise K.: Cost-effective and high-throughput merge network: Architecture for the fastest FPGA sorting accelerator. *ACM SIGARCH Computer Architecture News 44*, 4 (2017), 8–13.

[MW06] Mahovsky J., Wyvill B.: Memory-conserving bounding volume hierarchies with coherent raytracing. In *Computer Graphics Forum* (2006), vol. 25, pp. 173–182.

[ND12] Novák J., Dachsbacher C.: Rasterized bounding volume hierarchies. *Computer Graphics Forum 31*, 2pt2 (2012), 403–412.

[NKK*14] Nah J.-H., Kwon H., Kim D., Jeong C., Park J., Han T., Manocha D., Park W.: RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics 33*, 5 (2014), 162:1–15.

[NKP*15] Nah J.-H., Kim J., Park J., Lee W., Park J., Jung S., Park W., Manocha D., Han T.: HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (2015), 389–401.

[NPK*10] Nah J.-H., Park J.-S., Kim J.-W., Park C., Han T.-D.: Ordered depth-first layouts for ray tracing. In *SIGGRAPH Asia Sketches* (2010), p. 55.

[NPP*11] Nah J.-H., Park J.-S., Park C., Kim J.-W., Jung Y.-H., Park W.-C., Han T.-D.: T&I engine: Traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics 30*, 6 (2011), 160.

[ODJ16] Ogaki S., Derouet-Jourdan A.: An N-ary BVH child node sorting technique for occlusion test. *Journal of Computer Graphics Techniques 5*, 2 (2016), 22–37.

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002), 703–712.

[Pin10] Pinto A. S.: Adaptive collapsing on bounding volume hierarchies for ray-tracing. In *Proc. Eurographics (Short Papers)* (2010), pp. 73–76.

[PJB13] Pohl D., Johnson G. S., Bolkart T.: Improved pre-warping for wide angle, head mounted displays. In *Proc. ACM Symp. Virtual Reality Software and Technology* (2013), pp. 259–262.

[PJH16] Pharr M., Jakob W., Humphreys G.: *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[PKGH97] Pharr M., Kolb C., Gershbein R., Hanrahan P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. SIGGRAPH* (1997), pp. 101–108.

[PL10] Pantaleoni J., Luebke D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High-Performance Graphics* (2010), pp. 87–95.

[Pow15] PowerVR: PowerVR ray tracing, 2015. Accessed Oct 18, 2017. URL: https://imgtec.com/legacy-cpu-cores/ray-tracing/.

[RGD09] Ramani K., Gribble C. P., Davis A.: Streamray: A stream filtering architecture for coherent ray tracing. In *ACM SIGPLAN Notices* (2009), vol. 44, pp. 325–336.

[RHAZ06] Raabe A., Hochgürtel S., Anlauf J., Zachmann G.: Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Proc. Conf. Design, Automation and Test in Europe: Designers' Forum* (2006), pp. 206–211.

[RSH05] Reshetov A., Soupikov A., Hurley J.: Multi-level ray tracing algorithm. *ACM Transactions on Graphics 24*, 3 (2005), 1176–1185.

[RTKPS16] Richter-Trummer T., Kalkofen D., Park J., Schmalstieg D.: Instant mixed reality lighting from casual scanning. In *IEEE Int. Symp. Mixed and Augmented Reality* (2016), pp. 27–36.

[SCPC15] Srivastava A., Chen R., Prasanna V. K., Chelmis C.: A hybrid design for high performance large-scale sorting on FPGA. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs* (2015), pp. 1–6.

[SE10] Segovia B., Ernst M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proc. Graphics Interface* (2010), pp. 153–160.

[SFD09] Stich M., Friedrich H., Dietrich A.: Spatial splits in bounding volume hierarchies. In *Proc. High-Performance Graphics* (2009), pp. 7–13.

[SGK*17] Shkurko K., Grant T., Kopta D., Mallett I., Yuksel C., Brunvand E.: Dual streaming for hardware-accelerated ray tracing. In *Proc. High-Performance Graphics* (2017), p. 12.

[SKBD12] Spjut J., Kopta D., Brunvand E., Davis A.: A mobile accelerator architecture for ray tracing. In *Proc. Workshop on SoCs, Heterogeneous Architectures and Workloads* (2012).

[SKKB09] Spjut J., Kensler A., Kopta D., Brunvand E.: TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28*, 12 (2009), 1802–1815.

[SLM*16] Selgrad K., Lier A., Martinek M., Buchenau C., Guthe M., Kranz F., Schäfer H., Stamminger M.: A compressed representation for ray tracing parametric surfaces. *ACM Transactions on Graphics 36*, 1 (2016), 5.

[SMD*06] Stoll G., Mark W. R., Djeu P., Wang R., Elhassan I.: *Razor: An architecture for dynamic multiresolution ray tracing.* Tech. rep., University of Texas at Austin, 2006.

[Smi87] Smith A. J.: Line (block) size choice for CPU cache memories. *IEEE Transactions on Computers 100*, 9 (1987), 1063–1075.

[SSKN07] Shevtsov M., Soupikov A., Kapustin A., Novorod N.: Ray-triangle intersection algorithm for modern CPU architectures. In *Proc. GraphiCon* (2007), vol. 2007, pp. 33–39.

[SWS02] Schmittler J., Wald I., Slusallek P.: SaarCOR: a hardware architecture for ray tracing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware* (2002), pp. 27–36.

[Tay12] Taylor M. B.: Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. Design Automation Conf.* (2012), pp. 1131–1136.

[UVCK16] Usui T., Van Chu T., Kise K.: A cost-effective and scalable merge sorter tree on FPGAs. In *Proc. Int. Symp. Computing and Networking* (2016), pp. 47–56.

[VAMS16] Vaidyanathan K., Akenine-Möller T., Salvi M.: Watertight ray traversal with reduced precision. *Proc. High-Performance Graphics* (2016).

[VHB14] Vinkler M., Havran V., Bittner J.: Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In *Proc. Spring Conf. Computer Graphics* (2014), pp. 29–36.

[vKP10] van Krevelen D. W. F., Poelman R.: A survey of augmented reality technologies, applications and limitations. *International Journal of Virtual Reality*, 9 (2010), 220.

[Wal04] Wald I.: *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Saarland University, Germany, 2004.

[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2007), pp. 33–40.

[WBB08] Wald I., Benthin C., Boulos S.: Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2008), pp. 49–57.

[WBKP08] Walter B., Bala K., Kulkarni M., Pingali K.: Fast agglomerative clustering for rendering. In *Proc. IEEE Symp. Interactive Ray Tracing* (2008), pp. 81–86.

[WBS03] Wald I., Benthin C., Slusallek P.: Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* (2003), IEEE Computer Society, p. 11.

[WBS06] Woop S., Brunvand E., Slusallek P.: Estimating performance of a ray-tracing ASIC design. In *Proc. IEEE Symp. Interactive Ray Tracing* (2006), pp. 7–14.

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6.

[WIP08] Wald I., Ize T., Parker S. G.: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1 (2008), 3–13.

[WK06] Wächter C., Keller A.: Instant ray tracing: The bounding interval hierarchy. In *Proc. Eurographics Conf. Rendering Techniques* (2006), pp. 139–149.

[WMG*09] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722.

[WMS06] Woop S., Marmitt G., Slusallek P.: B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware* (2006), pp. 67–77.

[WRK*16] Weier M., Roth T., Kruijff E., Hinkenjann A., Pérard-Gayot A., Slusallek P., Li Y.: Foveated real-time ray tracing for head-mounted displays. *Computer Graphics Forum 35*, 7 (2016), 289–298.

[WSBW01] Wald I., Slusallek P., Benthin C., Wagner M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–165.

[WSS05] Woop S., Schmittler J., Slusallek P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444.

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014), 143.

[YCK*09] Yu I., Cox A., Kim M. H., Ritschel T., Grosch T., Dachsbacher C., Kautz J.: Perceptual influence of approximate visibility in indirect illumination. *ACM Transactions on Applied Perception 6*, 4 (2009), 24.

[YCM07] Yoon S.-E., Curtis S., Manocha D.: Ray tracing dynamic scenes using selective restructuring. In *Proc. Eurographics Conf. Rendering Techniques* (2007), pp. 73–84.

[YKL17] Ylitie H., Karras T., Laine S.: Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proc. High-Performance Graphics* (2017), p. 4.

[YL14] Yin M., Li S.: Fast BVH construction and refit for ray tracing of dynamic scenes. *Multimedia Tools and Applications 72*, 2 (2014), 1823–1839.

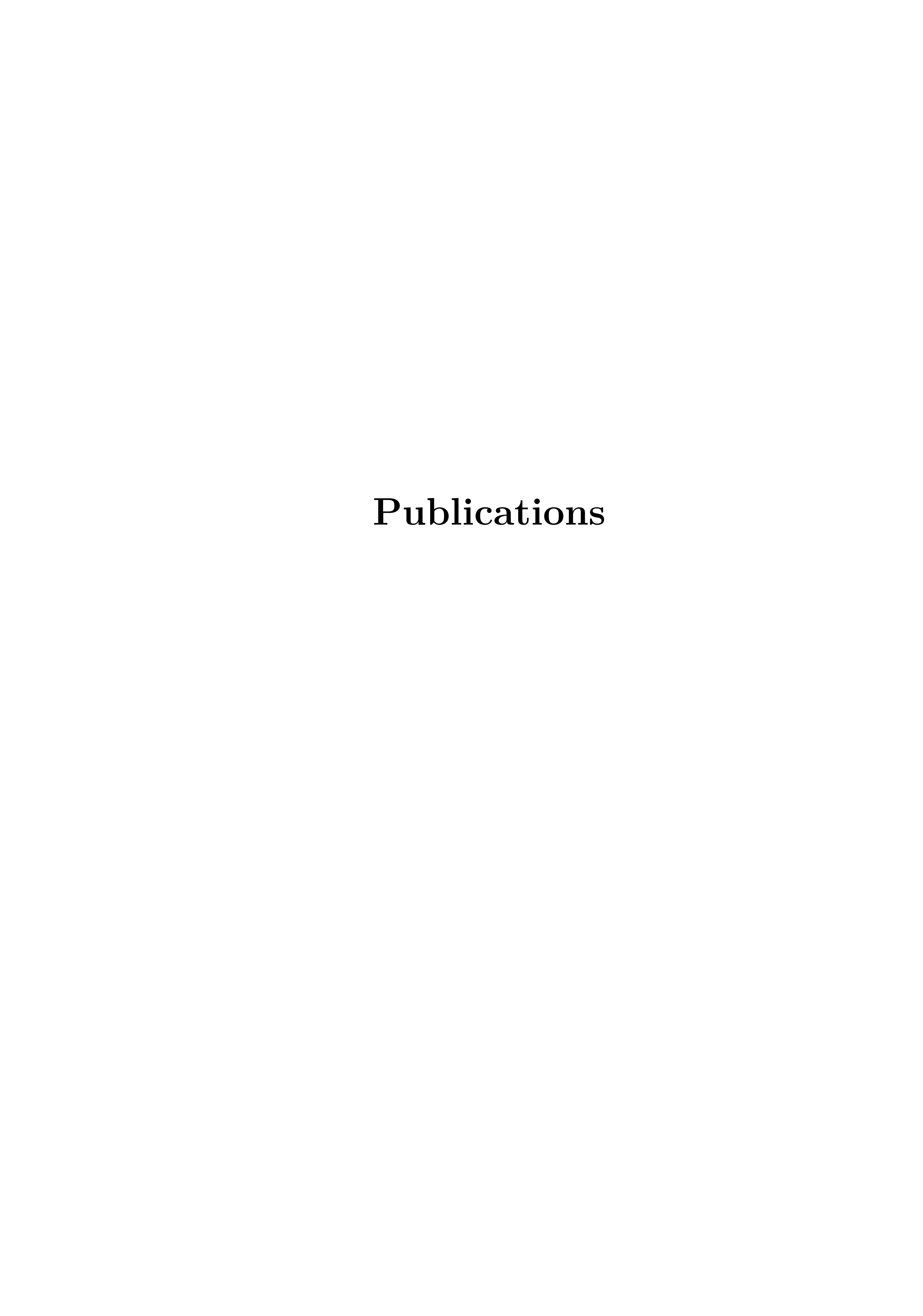
[YM06] Yoon S.-E., Manocha D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516.

[ZCP16] Zhang C., Chen R., Prasanna V.: High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshops* (2016), pp. 148–155.

[ZJL*15] Zwicker M., Jarosz W., Lehtinen J., Moon B., Ramamoorthi R., Rousselle F., Sen P., Soler C., Yoon S.-E.: Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum 34*, 2 (2015), 667–681.

[ZNA15] Zeidan M., Nazmy T., Aref M.: GPU-based out-of-core HLBVH construction. In *Proc. Eurographics Symposium on Rendering - Experimental Ideas & Implementations* (2015).

[ZU06] Zuniga M. R., Uhlmann J. K.: Ray queries with wide object isolation and the de-tree. *Journal of Graphics Tools 11*, 3 (2006), 27–45.



Publications

[P1] Publication 1

[P1] Viitanen T., Koskela M., Jääskeläinen P., Kultala H., Takala J.:
MergeTree: a HLBVH constructor for mobile systems.
In *SIGGRAPH Asia Technical Briefs* (2015), p. 12.

MergeTree: A HLBVH Constructor for Mobile Systems

Timo Viitanen Matias Koskela Pekka Jääskeläinen Heikki Kultala Jarmo Takala *

Tampere University of Technology, Finland

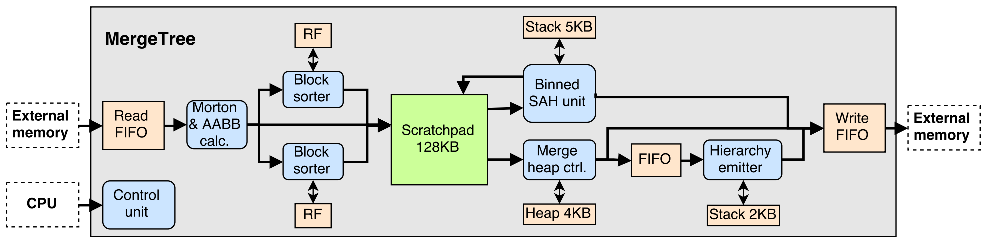


Figure 1: *Proposed hardware architecture. RF: Register File. SAH: Surface Area Heuristic.*

Abstract

Powerful hardware accelerators have been recently developed that put interactive ray-tracing even in the reach of mobile devices. However, supplying the rendering unit with up-to date acceleration trees remains difficult, so the rendered scenes are mostly static. The restricted memory bandwidth of a mobile device is a challenge with applying GPU-based tree construction algorithms. This paper describes MergeTree, a BVH tree constructor architecture based on the HLBVH algorithm, whose main features of interest are a streaming hierarchy emitter, an external sorting algorithm with provably minimal memory usage, and a hardware priority queue used to accelerate the external sort. In simulations, the resulting unit is faster by a factor of three than the state-of-the art hardware builder based on the binned SAH sweep algorithm.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray-tracing, ray-tracing hardware, bounding volume hierarchy, BVH, HLBVH

1 Introduction

Ray-tracing is a promising rendering technique for mobile systems. The runtime of the algorithm scales more with the number of pixels than the number of drawn primitives, making it ideal for small displays. Effects such as shadows, reflection and global illumination are more natural to express than in traditional rasterization based architectures. In proposed augmented reality applications physically based ray-tracing would seamlessly overlay virtual objects on a real environment. In recent years, increasingly powerful mobile ray-tracing accelerators have been developed [Lee et al. 2013; Nah et al. 2014]. However, these units have mostly been reliant on a prebuilt acceleration datastructure, restricting them to static scenes.

Currently the fastest GPU tree builders such as HLBVH [Garanzha et al. 2011] use a *Bounding Volume Hierarchy* (BVH) datastructure, and are based on sorting primitives according to the *morton codes* of their centroids, so this approach is interesting for a custom hardware unit. However, a direct adaptation of the GPU algorithms to a mobile context is hampered by memory bandwidth limitations: recent high-end mobile SoCs have an order-of-magnitude less memory bandwidth than high-end GPUs. Special techniques are necessary to cope with this environment, for example, mobile GPUs conserve bandwidth by means of tiling and texture compression.In this paper, we propose the first custom hardware architecture for HLBVH, which minimizes bandwidth usage through external sorting and streaming hierarchy emission. The architecture is evaluated by building a cycle-level simulator.

2 Related work

In the RayCore architecture [Nah et al. 2014], the scene geometry is split into two parts with separate acceleration trees: a small dynamic part is rebuilt on each frame using a hardware unit, and a larger static part is pre-built. Each ray traverses both trees to find the nearest intersection. The HART system [Nah et al. 2015] updates the BVH tree with hardware-accelerated *refit* operation instead of running a full rebuild on each frame. Since tree quality degrades with each refit, asynchronous rebuilds are run on the CPU to refresh it. Our design would be useful as a component in either system: in a refit-based renderer the faster asynchronous rebuilds would help the system adapt to sudden changes in the scene, while in a two-part system a larger dynamic part could be kept up to date.

Doyle et al. [2013] propose the first BVH construction hardware unit, which performs a binned *Surface Area Heuristic* (SAH) sweep. Our proposed design uses a similar but scaled-down unit as a subcomponent for the HLBVH top-level build stage.

The FastTree unit [Liu et al. 2015] uses Morton codes for k-d tree construction, and is the fastest k-d tree constructor hardware so far. They use a memory-intensive radix sort, but this does not appear to harm performance, since k-d trees are much more compute-intensive to construct than BVHs.

*e-mail:{timo.2.viitanen, matias.koskela, pekka.jaaskelainen, heikki.kultala, jarmo.takala}@tut.fi

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SA'15 Technical Briefs, November 02 – 06, 2015, Kobe, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3830-8/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2820903.2820916

3 Algorithm

Table 1: *Sorting algorithm comparison in tree construction, 32B IO words / primitive.*

	Prim. read	Sorting	BVH write	total
Sort Morton codes (8B per element)				
Radix-16 counting sort	10	48	24	82
Multimerge, 1 pass	10	4	24	38
Multimerge, 2 pass	10	8	24	42
Sort AABBs (28B per element)				
Radix-16 counting sort	10	168	14	192
Multimerge, 1 pass	10	14	14	38
Multimerge, 2 pass	10	28	14	52

The LBVH algorithm [Lauterbach et al. 2009] produces BVH trees by sorting the input primitives according to their Morton codes. After this, a BVH hierarchy can be emitted by binning primitives based on the bits of their Morton codes: e.g., the highest level split is generated, by placing primitives with MSB 0 and 1 in the left and right children of the top node, respectively. Lower hierarchy levels correspond to lower bits. This process is called *hierarchy emission*. The HLBVH algorithm [Garanzha et al. 2011] improves tree quality by generating the top levels of the tree with the slower binned SAH sweep algorithm.

Sorting accounts for much of the memory traffic in HLBVH, so we attempt to optimize it by referring to literature on *external sorting* data on slow magnetic disc drives. One optimal sorting algorithm in this environment is the *multimergesort* [Aggarwal and Vitter 1988]. Given N data elements that reside in slow external memory, a fast local memory of size M, and a preferred read length of B, the multimergesort first performs partial sorts for N/M M-sized blocks. After this, the algorithm runs multimerge passes which merge M/B sorted blocks into a larger block. Table 1 compares the minimum memory accesses of multimerge sorting in the context of tree construction to a typical radix-16 sort which takes eight passes through the data. If the scene can be processed in one pass, the multimerge sort uses less memory by a factor of two.

In addition to the choice of algorithm, we must also choose whether to sort the original primitives (40 bytes per item), their *Axis-Aligned Bounding Boxes* (AABBs) (28 bytes), or Morton code-reference pairs (8 bytes). At first Morton code sorting appears optimal, but it has the drawback that, when generating the hierarchy, we need to random access every triangle in the scene to generate AABBs. Table 1 shows that for a 1-pass multimerge sort, AABB sorting uses as much memory bandwidth as Morton code sorting, and the memory accesses are split more evenly between different stages of the algorithm. Therefore, we focus on AABB sorting in this work. However, this causes some overhead for large scenes that require more than one pass.

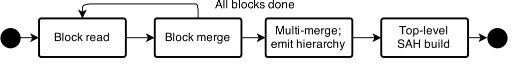


Figure 2: *Control state diagram.*

4 Architecture

The overall proposed MergeTree architecture is showin in Figure 1, and it operates according to the state diagram of Figure 2. The unit first performs $\lceil \frac{N}{M} \rceil$ partial sorts, which are split into *block read* and *block merge* states. Next, the sorted M-sized buffers are merged together, and the result is fed into a streaming hierarchy emitter, which produces BVH nodes. Finally, a high-level hierarchy is built with a separate binned SAH sweep unit.

4.1 Primitive input

We internally store data elements as AABBs augmented with a memory reference and a Morton code, for a total of 256 bits. The Morton codes are computed on the fly when reading in data, and omitted when writing out to memory.

4.2 Multi-merge hardware

A standard software implementation of the multimergesort algorithm places merge candidate values from each buffer into a heap datastructure. On each iteration, a minimum value is taken from the top of the heap, and the next value from the same block is inserted. A sequential heap implementation takes $\log n$ compare-swaps to insert an element, which is too slow. Fortunately, the process can be pipelined by inserting separate memory blocks and control hardware for each level of the heap. A similar hardware structure was proposed by Moon et al. [2000] to implement a hardware priority queue for networking hardware: we refer to their paper for a full discussion of the design tradeoffs.

It is difficult to fit any logic on the same clock cycle with the access delay of a large scratchpad, so we have the merge hardware output AABBs once per two cycles: one cycle is reserved for the memory access, and the other is used to perform the top-level heap insertion, which produces the next read address. The heap insertion cycle can be used to move data from the read queue into the scratchpad. This allows high clock frequencies to be reached, and also fits well with the streaming hierarchy emitter whose straightforward implementation processes, on average, one input per two cycles. The scratchpad is split into at least two banks, to allow concurrent reads of the selected AABB (for hierarchy emission) and the next AABB (for heap insertion).

We schedule memory reads by means of *double buffering*: each block in the main memory is represented by two buffers on the scratchpad, and when one buffer has been processed, a read is queued to replace it. If two buffers are processed from the same block before replacement data arrives, the merge heap stalls. It would also be interesting to evaluate the other well-known read scheduling technique of *forecasting*, where a second heap stores the final element of each buffer, and predicts which buffer will have to be replaced first.

4.3 Streaming hierarchy emission

In order to minimize external memory traffic, we stream sorted AABBs to a hardware state machine which implements a serial LBVH hierarchy emission algorithm. Figure 3 shows a visual example of the algorithm in operation. The generated inner node topology corresponds to the shown Morton code bits. Each node is output when sufficient primitives have been read to determine its child bounding boxes, resulting in a bottom-up order. Stack entries represent inner nodes whose right child is unknown: they consist of a left child AABB and a hierarchy level. For example, the final read leaf in Figure 3 gives sufficient information to generate the last 3 inner nodes on cycles 6, 7 and 8. Nodes with identical Morton codes

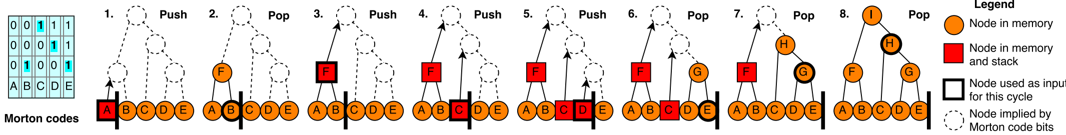


Figure 3: *Example of streaming hierarchy emission, which processes the given sequence of leaf nodes (A-E) and morton codes (left) to produce inner nodes (F-I). Hierarchy level of inner nodes is determined by (emphasized) highest differing bit of corresponding Morton codes.*

are handled by extracting differing bits from their indices, which generates a somewhat balanced subtree. Generating n inner nodes requires $2n$ stack operations. We assume the hardware can perform one stack operation per cycle. The full algorithm is as follows:

```
while True do
    input ← nextInput ;
    read nextInput from FIFO;
    diff ← highest diff. bit of input and nextInput ;
    while ¬ stack.empty() ∧ stack.top().diff < diff do
        BVHNode n(stack.pop().aabb, input) ;
        input ← n.aabb ;
        output[idx++] ← n ;
    end
    stack.push(input, diff) ;
end
```

4.4 Partial sort

The merge sorting hardware described above is straightforward to reuse for the scratchpad-sized partial sort, by configuring the merge heap so that each buffer on scratchpad is the final one is in its block, and no further buffers are fetched. Then the only additional work needed is to sort every buffer-sized subblock prior to merging. This is easy to implement concurrently with data reading, by streaming the read data into a small number of buffer-sized subblock sorters, which use a state machine to perform an insertion sort at a rate of one compare-and-swap per second.

4.5 Top-level SAH build

The HLBVH algorithm improves tree quality by constructing the highest levels of the tree with a binned SAH sweep. This can be performed with the hardware unit proposed by Doyle et al. [2013], but since the input size is small, a scaled-down version could be used with fewer computational resources. To evaluate the design, we use 8 bins, one parallel worker, and six pipelines for split AABB generation from bin AABBs, SAH computation, and split plane computation. For evaluation, we assume conservatively that the unit takes 32 cycles to do these tasks after processing a range of primitives, but it is likely that an optimized design can interleave some of this computation with another processing sweep.

5 Evaluation

Table 2: *Memory traffic comparison (MB).*

	[Doyle et al. 2013]	Proposed
Cloth (92K)	25	15
Conference (331K)	120	53
Dragon (871K)	380	140

In order to evaluate the architecture, we developed a cycle-level C++ simulator. The main components are simulated with cycle-accurate state machines. We model the external memory using the GDDR3 memory controller model from GPGPUSim, configured as 1GHz, 32-bit, dual channel, which is close to LPDDR3 in recent SoCs [Lee et al. 2013]. We assume an operating frequency of 1GHz, which we think is realistic at least in a recent process technology. The parameters of the hardware unit are set at $M = 4096$, $B = 8$, resulting in a unit with a 128KB scratchpad memory, which handles up to 1M triangles in one pass, and performs a 256-way merge. We include four block sorters for scalability. For comparison, Liu et al. [2015] have a 172KB scratchpad and Doyle et al. [2013] use 432KB.

The simulator was run on five test scenes, and the resulting trees were verified in a software ray-tracer. In Tables 3 and 2, the performance and memory traffic are compared to related work. Our construction speed is ca. 3 times faster than previous BVH hardware, 5 times faster than k-d tree hardware, and catches up with a GPU implementation of HLBVH except for the largest Dragon scene, though we are still behind the latest GPU implementation of LBVH. The speedup can be attributed to our choice of sorting algorithm: as shown in Table 1, a HLBVH based on a conventional radix sort would use 2× more memory bandwidth and, therefore, construction time. Figure 4 shows an example run of the simulator, using the small *sibenik* scene for clarity. The different execution states are visible: first the the unit runs block sorts which alternate between utilizing the subblock sorters and the merge heap. Most of the execution time is spent on the multimerge phase, which is clearly memory-limited: the merge heap runs at less than one-third of maximum capacity. Finally, the top-level SAH build is more compute-intensive and uses little memory. The unit utilized 82% of theoretical maximum bandwidth in this scene, and an average of 87% across all scenes, where the SAH build is less significant. Much of the idle time was due to switching between reads and writes.

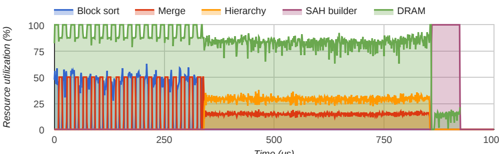


Figure 4: *Simulation trace for the sibenik scene.*

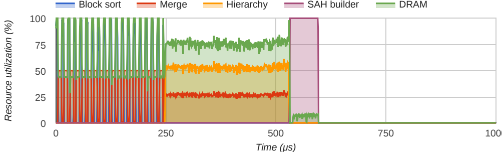


Figure 5: *Simulation trace with memory bus doubled to 32GB/s.*

Table 3: *Performance comparison: build times in milliseconds.*

	GTX 480 HLBVH [Garanzha et al. 2011]	GTX 480 LBVH [Karras 2012]	Hardware k-d tree [Liu et al. 2015]	Hardware SAH BVH [Doyle et al. 2013]	Hardware HLBVH (proposed)	
Mem. BW (GB/s)	133.9	133.9	16	36	16	32
Sibenik (75K)	-	-	6.6	-	0.92	0.60 (-35%)
Cloth (92K)	4.8	-	-	3	1.02	0.62 (-39%)
Fairy (174K)	-	0.99	10.8	-	1.98	1.24 (-37%)
Conference (331K)	6.2	1.45	17.2	11	3.79	2.54 (-33%)
Dragon (871K)	8.1	3.64	52.2	30	9.74	5.93 (-39%)

We performed chip area estimation using the same methodology as [Doyle et al. 2013; Liu et al. 2015]. The results indicate an area of $5.6mm^2$ at 65nm and $1.4mm^2$ when scaled to 28nm as in Liu et al. The proposed unit is expected to have low power consumption even compared to previous custom hardware since it performs fewer external memory accesses as shown in Table 2. The quality of the produced trees was estimated by computing their SAH and comparing against a binned SAH sweep with 16 splits. The HLBVH trees were 7% worse on average. The top-level SAH sweep improved estimated rendering performance by an average of 11% and took an average of 2% of the runtime, so it appears to be a good tradeoff except for very small scenes. We investigated how well the design scales to near-future mobile memory buses by rerunning the simulations with a doubled bus clock rate. As shown in Table 3, the build time decreased on average by 37%. Figure 5 shows the example *sibenik* simulator run repeated with the doubled bandwidth. The main scaling bottleneck is the block merge state, where the maximum output of the merge hardware is insufficient to saturate the bus. A possible optimization is to use double buffering, i.e., read data to one half of the scratchpad while merging the other half.

6 Limitations

One difficulty in the proposed design is handling scenes of over $\frac{M^2}{2B}$ primitives (1M with the evaluation setup), as they require more than one multimerge pass. It is simple to add control logic for multiple passes, but the AABB-sorting algorithm is then suboptimal. Another possibility is to enlarge the scratchpad: doubling the memory size M quadruples the model size that can be processed in one pass. In our experience at least a 512KB scratchpad memory can run at 1GHz; this would be sufficient for scenes of 16M triangles. We also have a built-in assumption that the application software can supply at least approximate scene bounds for generating efficient Morton codes: otherwise they need a separate pass through the inputs, increasing memory accesses by ca. 26%.

7 Conclusion and Future Work

We described a HLBVH-based BVH constructor architecture optimized for use in mobile devices. The unit is three times faster than the state of the art, showing that although HLBVH is associated with massively parallel GPU implementations, it is also suitable for a fast serial hardware design. The memory usage of the unit is close to a theoretical lower bound for a sorting-based tree construction, and simulations indicate that it runs fast enough to saturate current mobile memory buses, therefore, the build performance is close to maximum achievable in current mobile systems for this type of algorithm. The unit was evaluated as a standalone builder, but it could also be used as part of a refit-based system such as [Nah et al. 2015]. We are in the early stages of building an FPGA prototype, and are interested in also performing CMOS synthesis and place&route for more accurate area and power estimates.

Acknowledgements

This research is supported by the TUT doctoral program, Finnish Funding Agency for Technology and Innovation (project Parallel Acceleration 2, funding decision 40081/14), and ARTEMIS JU under grant agreement no 621439 (ALMARVI).

References

AGGARWAL, A., AND VITTER, J. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9, 1116–1127.

DOYLE, M., FOWLER, C., AND MANZKE, M. 2013. A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics 32*, 4, 139.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, 59–64.

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, Eurographics Association, 33–37.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, vol. 28, Wiley Online Library, 375–384.

LEE, W., SHIN, Y., LEE, J., KIM, J., NAH, J., JUNG, S., LEE, S., PARK, H., AND HAN, T. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, 109–119.

LIU, X., DENG, Y., NI, Y., AND LIL, Z. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 1595–1598.

MOON, S.-W., REXFORD, J., AND SHIN, K. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers 49*, 11, 1215–1227.

NAH, J., KWON, H., KIM, D., JEONG, C., PARK, J., HAN, T., MANOCHA, D., AND PARK, W. 2014. RayCore: a ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics 33*, 5, 132.

NAH, J., KIM, J., PARK, J., LEE, W., PARK, J., JUNG, S., PARK, W., MANOCHA, D., AND HAN, T. 2015. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3, 389–401.

[P2] Publication 2

[P2] Viitanen T., Koskela M., Jääskeläinen P., Kultala H., Takala J.: MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing. *ACM Transactions on Graphics 36*, 5 (2017), 169.

MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing

TIMO VIITANEN, MATIAS KOSKELA, PEKKA JÄÄSKELÄINEN, HEIKKI KULTALA, and JARMO TAKALA
Tampere University of Technology, Finland

Ray tracing is a computationally intensive rendering technique traditionally used in offline high-quality rendering. Powerful hardware accelerators have been recently developed that put real-time ray tracing even in the reach of mobile devices. However, rendering animated scenes remains difficult, as updating the acceleration trees for each frame is a memory-intensive process. This article proposes MergeTree, the first hardware architecture for *Hierarchical Linear Bounding Volume Hierarchy* (HLBVH) construction, designed to minimize memory traffic. For evaluation, the hardware constructor is synthesized on a 28nm process technology. Compared to a state-of-the-art binned surface area heuristic sweep (SAH) builder, the present work speeds up construction by a factor of 5, reduces build energy by a factor of 3.2, and memory traffic by a factor of 3. A software HLBVH builder on a graphics processing unit (GPU) requires 3.3 times more memory traffic. To take tree quality into account, a rendering accelerator is modeled alongside the builder. Given the use of a toplevel build to improve tree quality, the proposed builder reduces system energy per frame by an average 41% with primary rays and 13% with diffuse rays. In large (>500K triangles) scenes, the difference is more pronounced, 62% and 35%, respectively.

CCS Concepts: • **Computing methodologies → Ray tracing**; Graphics processors;

Additional Key Words and Phrases: Ray tracing, ray-tracing hardware, bounding volume hierarchy, BVH, HLBVH

ACM Reference Format:

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2017. MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing. ACM Trans. Graph. 36, 5, Article 169 (October 2017), 14 pages.
DOI: http://dx.doi.org/10.1145/3132702

1. INTRODUCTION

Ray tracing is a rendering technique where effects such as shadows, reflection, and global illumination are more natural to express than in rasterization.

Authors' addresses: T. Viitanen, M. Koskela, P. Jääskeläinen, H. Kultala, and J. Takala; emails: {timo.2.viitanen, matias.koskela, pekka.jaaskelainen, heikki.kultala, jarmo.takala}@tut.fi.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

0730-0301/2017/10-ART169 $15.00
DOI: http://dx.doi.org/10.1145/3132702

Mainstream use of ray tracing has been restricted to offline rendering, but recent years have seen a concerted effort by the academia and the industry to enable real-time ray tracing of dynamic scenes. One prong of this effort has been the development of dedicated ray-tracing hardware architectures, both based on programmable processors [47, 48, 53], fixed-function hardware pipelines [32, 41], reconfigurable pipelines [28], and building on conventional, rasterization-based GPUs [25]. Several works focus on mobile systems, reasoning that the ray-tracing approach scales well to drawing complex scenes on small displays [48], or aiming to create mobile augmented-reality experiences with physically based lighting [32]. Recently, a commercial mobile GPU IP with ray-tracing acceleration has been announced alongside a programming API [44].

Recent ray-tracing hardware accelerators are able enable real-time ray tracing, so far restricted to high-end desktop GPUs, on mobile devices. However, they have so far been largely restricted to scenes with little or no animated content. The reason is that fast rendering algorithms require the scene to be organized in an acceleration data structure such as a *Bounding Volume Hierarchy* (BVH) tree. When displaying a dynamic scene, the data structure needs to be updated or rebuilt on each frame as the scene changes, posing an additional computational challenge. Given enough animated geometry, construction effort overtakes rendering, as ray tracing scales logarithmically with the number of scene primitives, while construction algorithms have $O(n)$ [30] or $O(n \log n)$ [51] complexity. On the desktop, tree construction has been the subject of intensive research, and fast GPU tree builders now exist that organize large scenes at real-time rates. The fastest builders are based on the *Linear Bounding Volume Hierarchy* (LBVH) algorithm by Lauterbach et al. [30], and the improved *Hierarchical LBVH* (HLBVH) by Pantaleoni and Luebke [42], and are able to organize scenes with millions of triangles in real time. These builders leverage the massive amount of computing resources and memory bandwidth available on desktop GPUs, and are, therefore, not directly applicable on mobile systems with limited resources.

A particular restriction of mobile devices is their limited memory bandwidth: A high-end mobile *System-on-Chip* (SoC) has an order of magnitude less memory bandwidth than a high-end desktop GPU. CMOS logic scaling has allowed increasingly complex on-chip computation to fit in the tight power budgets of mobile SoCs, but the energy cost of off-chip communication has scaled down at a slower pace and is now very expensive compared to computation. For example, reading the operands of a double-precision multiply-add from external memory and writing back the result costs ca. 200 times more energy than the arithmetic itself [24]. Hence, the design of mobile hardware is an exercise of minimizing memory accesses to work around the memory bottleneck. Mobile GPUs incorporate a slew of special architectural techniques to this end, such as tile-based rendering, texture compression [5], and frame buffer compression [46]. A similar body of memory-conserving techiques is emerging for ray-tracing accelerators, including treelet scheduling [3], streaming data models [28], and quantized trees [25].

Tree construction for ray tracing is even more memory intensive than rendering, as the fast sorting-based build algorithms iterate over datasets of hundreds of megabytes and perform little computation for each element.

This article focuses on mobile hardware acceleration of the HLBVH algorithm [42]. HLBVH is interesting as a powerful builder in its own right and as a component in virtually all build algorithms that aim for a fast build time [12, 18, 23]. HLBVH is also an interesting target for hardware acceleration, since it does not make heavy use of floating-point arithmetic, hence, a hardware builder could fit in a small silicon footprint. We investigate whether the high build performance of HLBVH on GPU can translate into energy-efficient operation in a mobile context.

The main contributions of this article are as follows. We propose the first hardware HLBVH builder architecture, named MergeTree. MergeTree incorporates novel architectural techniques to reduce memory bandwidth usage: a hardware-accelerated *external sort* and a novel streaming algorithm for joint hierarchy emission and AABB computation, which operates directly on the sort outputs. The proposed architecture is evaluated via logic synthesis and power analysis on a 28nm *Fully Depleted Silicon on Insulator* (FD-SOI) process technology and by means of a system-level model that includes traversal and intersection hardware. In addition, two toplevel builds are evaluated as inexpensive postprocessing steps to improve tree quality. Early simulation results for MergeTree were reported in a conference brief [49].

Compared to previous work that uses the more expensive binned SAH algorithm [13], MergeTree gives large improvements in build performance, energy efficiency, memory traffic, and silicon area at the cost of reduced tree quality. Toplevel builds are able to recover much of the quality at low cost. System-level modeling shows that with large animated scenes, the energy cost of tree construction becomes comparable to the cost of rendering and takes up a significant fraction of a mobile power budget. Hence, the build energy savings from MergeTree translate into significant system energy savings despite the slightly lower tree quality. We also observe that most of the energy footprint in hardware-accelerated tree construction is due to DRAM traffic. A direct translation of GPU HLBVH algorithms to hardware, without the proposed memory traffic optimizations, would have energy consumption and runtime similarly to those in Reference [13].

This article is organized as follows. Section 2 discusses background on BVH construction algorithms. Section 3 reviews related work on hardware tree builders and sorting units. Section 4 discusses the basic algorithmic approach in this work and tradeoffs, while Section 5 describes the hardware architecture implementing the chosen algorithm. In Section 6, the architecture is evaluated by means of ASIC synthesis and system-level simulations, and a detailed power analysis is presented. Section 7 discusses limitations of the proposed architecture and future work, and Section 8 concludes the article.

2. PRELIMINARIES

In a BVH, each node subdivides primitives into two disjoint sets, whose *Axis-Aligned Bounding Boxes* (AABB) are stored. If a traced ray does not intersect an AABB, then all primitives underneath can be discarded, greatly speeding up the rendering process. A standard way to evaluate the quality of a BVH tree is its *Surface Area Heuristic* (SAH) cost, introduced by Goldsmith and Salmon [19]. The SAH cost of a data structure is the expected cost to traverse a random non-terminating ray through the scene.

A gold-standard way to construct BVH trees is the SAH sweep [36], a greedy top-down partitioning algorithm that at each step evaluates all possible axis-aligned splits of the primitives, according to their AABB centroids, into two subset. The algorithm then selects the split with the lowest SAH and repeats recursively for each subset. Since the basic SAH sweep has a long runtime, often the binned variation [51] is used instead, which evaluates only, for example, 8 or 16 possible splits per axis.

Starting with *Linear BVH* (LBVH) by Lauterbach et al. [30], a family of GPU construction algorithms has been proposed that are orders of magnitude faster than SAH-based builders. In LBVH, the scene primitives are first sorted according to the Morton codes of their AABB centroids, and then in the process of *hierarchy emission*, a BVH hierarchy is built that has a binary radix tree topology with regards to the sorted Morton codes. Finally, the AABBs of each node are computed in a bottom-up order.

Pantaleoni and Luebke [42] propose *Hierarchical Linear BVH* (HLBVH) with improved build performance and a more compact memory layout compared to LBVH. They further suggest improving tree quality by rebuilding upper levels of the tree with a binned SAH sweep, in an approach called HLBVH+SAH. We use this terminology in the present work, though several works use HLBVH to denote HLBVH+SAH. Garanzha et al. [17] and Karras [22] have further optimized the GPU software implementation of HLBVH, especially the hierarchy emission that is non-trivial to parallelize. Most recently, Apetrei [6] combined the hierarchy emission and AABB calculation stages into a single step for a further speedup.

More complex algorithms have been built around HLBVH that further improve tree quality. Karras and Aila [23] divide a HLBVH tree into treelets and rearrange nodes within each treelet in parallel to achieve higher tree quality, in an approach later dubbed *Treelet Restructuring BVH* (TRBVH). In the *Agglomerative TR-BVH* (ATRBVH) of Domingues and Pedrini [12], the exhaustive search of treelet permutations in TRBVH is replaced with an agglomerative build, yielding nearly the same tree quality at a fraction of the build time. Garanzha et al. [18] sort primitives according to their Morton codes, but instead of the HLBVH hierarchy emission, they store primitive counts in a multi-level grid and perform an approximate SAH sweep. Both TRBVH and Garangzha et al. [18] also break large triangles with spatial splits to improve tree quality. Recently, Ganestam and Doggett [16] proposed a fast, high-quality preprocessing step for triangle splitting.

3. RELATED WORK

In this section, we introduce related work on hardware architectures for tree construction and sorting.

3.1 Tree Build and Update Hardware

Some hardware builders have been proposed for k-d trees, an alternative acceleration data structure to BVH. The RayCore architecture [39] incorporates a hardware k-d tree builder unit that builds high levels of the hierarchy with a binned SAH sweep, and switches to a sorting-based build when the dataset is small enough to fit in on-chip SRAM. The builder is suitable only for small models, for example, a 64K triangle scene already takes 0.1 seconds to build. Therefore, they also design their renderer to use two acceleration trees: the smaller tree contains all animated geometry and is rebuilt with the hardware unit. The FastTree unit [34] uses Morton codes for k-d tree construction, and is the fastest k-d tree constructor hardware so far, at ca. 4 times the performance of the RayCore builder.

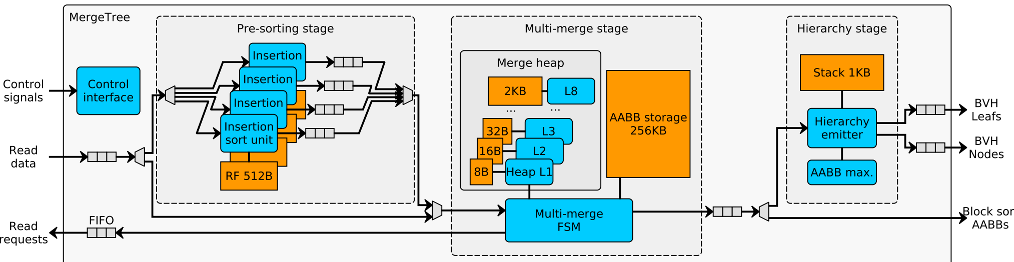


Fig. 1. The proposed hardware architecture, configured for single-pass tree building for scenes of up to 2M triangles. RF, Register File. BVH, Bounding Volume Hierarchy. AABB, Axis Aligned Bounding Box. FSM, Finite-State Machine. FIFO, First In First Out buffer.

However, tree quality is not evaluated. They use a memory-intensive radix sort, but this does not appear to harm performance, since k-d trees are much more compute-intensive to construct than BVHs, so their memory interface is not stressed.

In the literature, BVH trees are shown to be less expensive to build and update than k-d trees. Doyle et al. [13] propose the first hardware architecture for BVH construction, which implements the recursive binned SAH sweep algorithm. The architecture was recently prototyped on FPGA [14]. The HART rendering system [40] updates the BVH tree with hardware-accelerated *refit* operation instead of running a full rebuild on each frame. Since tree quality degrades with each refit, asynchronous rebuilds are run on the CPU to refresh the tree.

The present work is the first hardware implementation of the HLBVH algorithm, which is the basis for most high-performance GPU builders. In contrast to GPU implementations of HLBVH [6, 17, 22, 42], we adapt the algorithm for a streaming, hardware-oriented implementation with minimal memory traffic. The produced trees remain identical to the original work [42]. Our main point of comparison is the state-of-the-art builder by Doyle et al. [13], which implements binned SAH, a more computationally expensive algorithm. Compared to a refit accelerator [40], the proposed builder is able to handle animations that affect mesh topology, for example, fluids rendered with Marching Cubes, and can handle animation frames with entirely new geometry.

3.2 Sorting Hardware

The multi-merge sort approach has recently been used to sort large bodies of data with FPGAs to accelerate database operations. Koch and Torrensen [27] implement their multi-merge logic with a tree of comparators, and merge data from up to 102 input buffers. As a main difficulty in comparator tree design, they identify the problem of propagating back-pressure through the tree in a single cycle, and solve the issue by inserting decoupling FIFOs that split the tree into smaller sub-trees. Moreover, they pipeline the compare operations to get a higher operating frequency on FPGA. Casper et al. [9] further increase throughput by augmenting the top of the tree with comparators that produce multiple sorted values per cycle. They demonstrate merges from up to 8K input buffers per cycle, but in this case require over 2MB of buffer memories. The proposed accelerator implements a novel multi-merge based on a pipelined hardware heap rather than a comparator tree, giving a compact silicon area footprint at the cost of reduced throughput.

```
struct AABB {
        float lb_x, lb_y, lb_z;
        float ub_x, ub_y, ub_z;
        // 0 if child is an inner node.
        int leaf_size;
        // Index of inner node or leaf.
        int child_idx;
};

struct BVHNode {
        AABB aabbs[2];
};

BVHNode nodes[N];
int leafs[M];
```

Fig. 2. BVH data structure.

4. ALGORITHM DESIGN

In this section we describe how we adapt Pantaleoni and Luebke's [42] HLBVH algorithm to reduce memory traffic.

4.1 Data Structure Design

There are several variations of BVHs in the literature, and the chosen variant can have implications on build performance, so we describe the layout used in this work in detail here. In this layout, the node data structure stores the axis aligned bounding boxes of its two children and pointers to them. Áfra et al. [1] describe this arrangement as MBVH2. A leaf size field determines whether the child is another node or a leaf. Leafs are contiguous sub-arrays in a leaf table, bounded by the index and leaf size fields. Each entry in the table is a pointer to primitive data, which is used for ray-primitive tests and shading. The complete node data structure is 64 bytes long as shown in Figure 2.

There are many variations on the above details in the literature. Sometimes a node only has its own AABB and two child pointers, but the two-AABB structure is superior in hardware ray tracing [31]. More importantly, many high-performance ray tracers, for example. Aila and Laine [4], store primitive data in the leaf table, foregoing the extra indirection per rendering. Often the primitives are also preprocessed for faster intersection testing with, for example, Shevtsov's [45] method. Since primitives in the same leaf may not have been contiguous in the input data, this implies rearranging the primitives.

The present work opts to use a reference table for two main reasons. First, we try to make our results compatible with the main prior work by Doyle [13], which to our best understanding has this structure. Second, by taking primitive AABBs as input, the resulting

Table I. Sorting Algorithm Comparison for LBVH Construction

	Prim. read	Sort	BVH write	Total
Sort Morton codes				
Radix-16 counting sort	32	192	100	324
Multimergesort, 1 pass	32	16	100	148
Multimergesort, 2 passes	32	32	100	164
Sort AABBs				
Radix-16 counting sort	32	768	68	868
Multimergesort, 1 pass	32	64	68	164
Multimergesort, 2 passes	32	128	68	228

Note: External memory traffic in bytes.

architecture is generic to any primitive type for which an AABB can be computed - some examples of useful primitives are the pyramidal displacement mapped surfaces in Ref. [39] and indexed-vertex triangle lists in Ref. [25]. A primitive-leaf table could be produced as a post-processing step.

4.2 Sorting

Sorting accounts for much of the memory traffic in HLBVH, so we optimized it by referring to literature on *external sorting* data on slow magnetic disc drives. One optimal sorting algorithm in this environment is the *multimergesort* [2]. Given N data elements that reside in slow external memory, a fast local memory of size M, and a preferred read length of B, the multimergesort first performs partial sorts for N/M blocks of size M. After this, the algorithm runs multi-merge passes that merge M/B sorted blocks into a larger block. Table I compares the minimum memory accesses of multi-mergesorting in the context of tree construction to a typical radix-16 sort. Assuming one primitive per leaf, a BVH organizing N primitives has $N-1$ nodes. Any builder, then, has unavoidable memory traffic from primitive input (32B per input AABB) and hierarchy emission (64B per input AABB for nodes and 4B for the leaf table). In addition, primitive sorting requires memory accesses.

GPU implementations often use, for example, a radix-16 parallel prefix sort, which performs eight passes through the data, each pass reordering the data according to four bits of the sort key. In each pass the entire data array is read twice and written once. Assuming the sort operates on 8B Morton code - primitive reference pairs, it then requires $3 \times 8 \times 8\text{B} = 192\text{B}$ traffic per inpt AABB. Finally, the joint hierarchy emission and AABB computation stage must fetch the primitive AABBs referenced by the sort results, adding 32B to the unavoidable 68B output traffic. Out of the total traffic of $324B$, more than half is produced by the sort. Replacing the radix sort with multimergesort and assuming a small enough scene to sort in a single pass (2M triangles in the proposed design), the sort traffic drops to a negligible $16B$, with an additional $16B$ per pass. Assuming the AABBs from primitive input may be streamed on-chip to the block sort stage, and the results of the multi-merge to the hierarchy emission stage, the sorted pairs are only accessed twice, for $16B$ traffic. The total traffic of $148B$ is less than half that of the radix sort case.

The reads and writes in Table I are otherwise consequent, except when the hierarchy emission stage loads the primitive AABBs referenced by each sort output, it generates inefficient 32B random accesses. Consequently, it is interesting to directly sort the primitive AABBs instead of references. Direct AABB sorting is clearly inefficient with a radix sort due to the quadrupled sorting traffic. With multimergesort, the extra traffic is smaller (48B), and nearly offset by the removed hierarchy emission loads (32B). Total traffic still

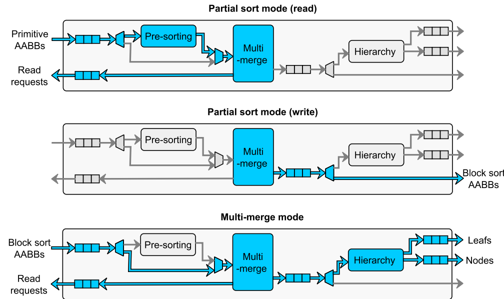


Fig. 3. Operating modes of the proposed architecture.

increases by ca. 11%, but all memory accesses can now be arranged in long consecutive bursts that are more efficient. If two or more multi-merge passes are needed, then the advantage of AABB sorting is less clear. It is, then, desirable to use AABBs as sorting elements and support as wide a merge as practical, so scenes of interest fit in a single pass. We use the AABB multi-merge approach as the basis for the present work. It should be mentioned that, in the extreme, the primitives themselves could be used as sort elements. It is inexpensive to add hardware to recompute their AABBs and Morton codes on demand, so the main cost would be increased memory traffic and on-chip storage. This approach is efficient for generating a leaf array with primitive data, discussed in the previous section, but foregoes genericity of the architecture for different primitives.

5. HARDWARE ARCHITECTURE

This section describes a hardware architecture named MergeTree that implements the designed construction algorithm. A block diagram of the architecture is shown in Figure 1. The architecture can be divided into the *pre-sorting* and *multi-merge* stages that sort an array of input AABBs according to their Morton codes, and a *hierarchy stage* that processes the sorted AABBs to emit a BVH tree. It has two operating modes where different sub-modules are active, as shown in Figure 3. The partial sort mode is used to generate $\frac{N}{M}$ AABB arrays small enough to fit in the on-chip *AABB storage*, while the hierarchy stage is inactive. In the multi-merge mode, the arrays are merged into a final sorted sequence and fed into the hierarchy stage. The multi-merge mode forms the backbone of the present architecture: It is a serial process that requires a specialized hardware pipeline for good performance. In contrast, the partial sorts parallelize well, and could be done with the multi-core CPU or mobile GPU in a SoC. The partial sorts are only integrated into the MergeTree to reuse hardware from the multi-merge mode. The following subsections first describe the multi-merge mode resources, and then the partial sorting scheme.

5.1 Primitive Input

The main data format used for input and internal storage is an AABB with three lower-bound and three upper-bound coordinates, a memory reference to primitive data, and a Morton code, for a total of 256 bits.

5.2 Heap Unit

The main component of the proposed algorithm that requires hardware acceleration is the multi-merge from many input sequences

into a single, sorted output stream. Hardware acceleration is necessary, since a sequential heap implementation for a heap of capacity n takes up to $\log n$ compare-swaps to insert an element, which is too slow for our use. Since comparator trees, used by recent sorting accelerators [9, 27], appear unreasonably large when scaled to wide inputs, we use an alternate approach of implementing a pipelined heap data structure in hardware. In software implementations the heap is stored in a single array in memory, where the children of an element can be found with simple arithmetic. However, in custom-designed hardware, each level of the heap can be implemented as a separate memory module, and heap operations would then operate in a pipelined manner, such that a new heap operation can be started while previous operations are still propagating toward deeper levels. This hardware structure was proposed by Bhagwan and Lin [8] for the implementation of large priority queues in telecommunications processors. Ioannou and Katevenis [20] optimize the design for clock speed by overlapping stages of computation. In this work, we largely follow the design of the latter work, and we refer the reader to their article [20] for details. We support the *insert* and *replace* operations, and implement *remove* as a *replace* with a large special-case value ∞. Replace operations have a maximum throughput of one value per two clock cycles, since fully pipelined operation would need an expensive set of global bypass wires between heap levels.

Detailed comparison to comparator trees is outside the scope of this article, but it should be noted that trees have $\mathcal{O}(n \log n)$ comparators with regards to the merge width, and the present design has $\mathcal{O}(\log n)$. The heap is, consequently, easier to scale to the wide merges desired in this work. Our throughput is lower than an optimized comparator tree, but since we have a much higher clock frequency available on ASIC than FPGA, and are sorting larger data elements, this is less of an issue than in the FPGA works. If more throughput is desired, then it may be interesting to use a hybrid scheme similar to that in Reference [9], combining a small toplevel comparator tree with subtrees implemented as heaps.

5.3 Multi-Merge Unit

The pipelined heap is connected to the rest of the system by a hardware finite-state machine that initializes the heap, requests replacement data from the memory and feeds it into the heap, and emits output data. To limit the size of the heap, the full primitive AABBs are stored in a SRAM scratchpad memory, and the heap only contains Morton codes and DRAM addresses of the corresponding primitives. The scratchpad memory is organized into a set of double-buffered queues for each block being multi-merged: When one buffer has been processed, a memory read is queued to replace it, but processing can continue from the other side of the double buffer. Only if the second half of the double buffer is also consumed before replacement input arrives for the first half, does the multi-merge unit need to stall. With double buffering, the multi-merge unit provides a degree of memory latency hiding, as the merge process is likely to visit multiple other buffers between the B elements of a single buffer. This property depends on the heap accesses being well distributed between the blocks: If, for example, the input data are already sorted, performance may suffer.

At the outset of partial sort and multi-merge modes, the FSM makes memory requests to fill the scratchpad and inserts the initial elements of each buffer into the empty heap. In steady-state operation, on each even cycle, the finite-state machine reads the top element of the heap, bit manipulates its DRAM address to find the scratchpad location of the following data element in the same block, and begins a SRAM read at that location. On the following cycle

Even cycle | **Odd cycle**

Fig. 4. Steady-state behavior of the multi-merge stage. morton(i): Morton code, addr(i): scratchpad address and AABB(i) AABB of the ith sorted element. After the heap outputs a sorted element (1), the control FSM first fetches its successor (X) from the scratchpad and performs a heap *replace operation*. Next, it fetches the original element (1) for output. The unit alternates between the two states shown: on even cycles a scratchpad read is initialized with an address read from the heap unit, on odd cycles data is fed from the scratchpad to the heap, and on the next even cycle, to the output FIFO.

the SRAM data are available and used for a *replace* heap operation. On odd cycles, the AABB corresponding to the previously read heap-top value is read from scratchpad and is written to the output FIFO on the next even cycle, simultaneously with the next heap-top read, in a pipelined manner. Like the heap unit, this design is also half-pipelined, producing one sorted AABB per two cycles in the absence of stalls.

Figure 4 demonstrates this steady-state behavior. On the example even cycle, the scratchpad address addr(1) of a sorted element is read from the heap, incremented, and used to initialize a scratchpad read of the element's successor in the same buffer. On the following odd cycle the successor's Morton code morton(X) is available from the scratchpad and is used to replace the top element of the heap. Simultaneously, a scratchpad read is initialized with the original (unincremented) address. On the next even cycle, the sorted element's AABB would be read from the scratchpad and written to the output, as is done for the previous sorted element (0) in Figure 4.

A separate register array tracks that buffers in the scratchpad are valid: If the finite-state machine attempts to fetch invalid data, execution instead stalls until the referenced data is available. Locations are invalidated when consumed, and validated again when replaced from memory. When the final element from the given block or scene is read, a *remove* operation is performed instead of a *replace*. To avoid special-case handling for the possible small buffer at the end of the scene, it is padded to full size with special-case AABBs with a higher Morton code than is generated for normal scene geometry: These are, then, sorted to the end of the scene and ignored by subsequent processing.

5.4 Hierarchy Emitter

To minimize external memory traffic, we stream sorted AABBs to a hardware finite-state machine that implements a serial HLBVH hierarchy emission algorithm. The hardware unit implements Algorithm 1, such that a single stack operation is performed per cycle.

Figure 5 shows a visual example of the algorithm in operation. The algorithm reads in a sorted stream of AABBs, and computes the highest differing bit between each pair of successive Morton

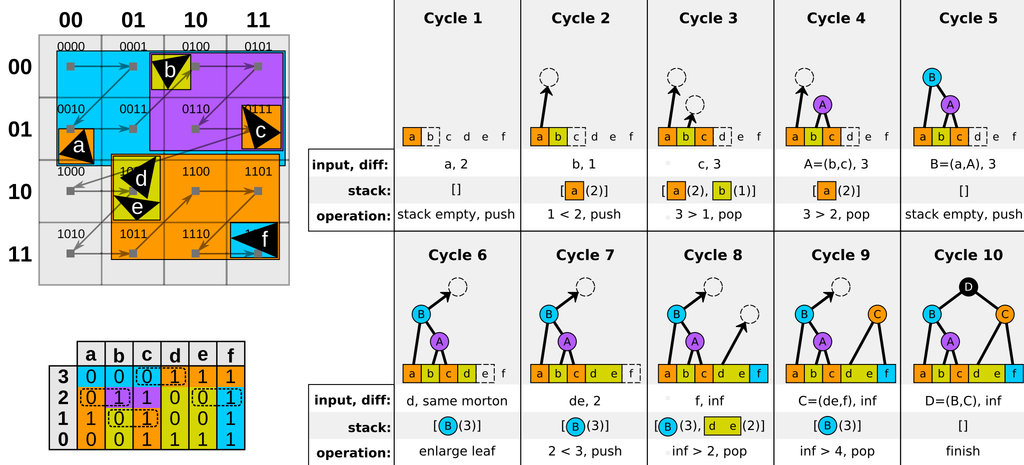


Fig. 5. Example of streaming hierarchy emission, which processes the given sequence of primitives ((a)–(f)) and their Morton codes (bottom left) to produce inner nodes ((A)–(D)) and leaf table entries for each primitive. Primitives, Morton curve and generated hierarchy are shown in the top left. *diff* is the highest differing bit in the morton codes of *input* and the following primitive, and determines the hierarchy level of the corresponding inner node. If *current* is an inner node, then it has the *diff* of the last primitive it contains: For example, inner node B in Cycle 5 contains primitives a, b, and c, and has the *diff* of c, 3.

ALGORITHM 1: Streaming hierarchy emission algorithm

```
1  while True do
2      input ← nextInput ;
3      read nextInput from FIFO;
4      while input and nextInput have the same Morton codes
       do
5          input ← nextInput ;
6          read nextInput from FIFO;
7          input ← combine ( input, nextInput );
8      end
9      diff ← highest diff. Morton bit of input and nextInput ;
10     while ¬ stack.empty() ∧ stack.top().diff < diff do
11         BVHNode n( stack.pop().aabb, input ) ;
12         input ← n.aabb ;
13         output.push( n ) ;
14     end
15     if diff > highlevel_threshold then
16         highlevel_output.push( input ) ;
17     else
18         stack.push( input, diff ) ;
19     end
20 end
```

codes, which is interpreted as a hierarchy level for an inner node to be generated. Based on each input's hierarchy level, the unit either concatenates the input into a large leaf (lines 4..8), pushes it into a small hardware stack, or combines it with the top AABB in the stack to generate a node. The latter process is then repeated with the AABB of the generated node used as the input node, until a higher-level element is found in the stack, or the stack is empty. Each node is output when sufficient primitives have been read to determine its child bounding boxes, resulting in a bottom-up, depth-first order. Stack entries represent inner nodes whose right child is unknown: They consist of a left child AABB and a hierarchy level. Optionally, a *highlevel_threshold* parameter can be supported by the hardware to emit high-level nodes for a separate toplevel build (lines 15..19), as in the HLBVH+SAH by Pantaleoni and Luebke [42]. The unit then generates separate trees for each highlevel grid cell, and outputs their AABBs to a buffer for postprocessing. When the parameter is greater than the highest Morton code bit, the unit's reverts to conventional HLBVH and outputs a single tree.

An example of the algorithm is shown in Figure 5. In Cycles 1 and 2 in Figure 5, stack entries for nodes on hierarchy levels 2 and 1 are pushed to the stack: The left child of the level-2 node is the primitive *a*, and child of the level-1 node, *b*. The next encountered hierarchy level of 3 is higher than the stack-top at level 1, so the stack-top node *A* can be completed. On Cycle 4, the next node in stack can be completed. On Cycle 5, a stack entry containing the entire subtree constructed so far, is pushed to the stack: It will become the root of the tree. Cycles 6 through 9 finish the right-side subtree in the same manner, except that primitives *d* and *e* have the same Morton code, and so are combined to the same leaf. Finally on Cycle 10, the top node is emitted. Cycles 8..10 also show that processing finishes with a special-case input AABB with higher Morton code value than in the rest of the geometry: This causes the remaining stack entries to be popped. The generated inner node topology in this example corresponds to the Morton code bits shown in Figure 5.

It is visible that generating n inner nodes requires $2n$ stack operations, while enlarging a leaf takes 1 cycle. Since a BVH organizing m leafs has at most $m - 1$ nodes, the worst-case runtime of the emitter is $2m - 2$ cycles for for m inputs, when there is exactly one primitive per leaf. The average throughput of the hierarchy emitter is, then, the same or higher as that of the multi-merge unit, but data are consumed at an uneven rate depending on inputs, so

a FIFO buffer is required between the two units. Hierarchy emission finishes by assuming the highest differing bit after the final sorted primitive is ∞, which causes all remaining stack entries to be popped. The maximum required stack depth is the same as the number of Morton code bits, that is, possible hierarchy levels.

5.5 Partial Sort

The multi-merge stage described above is straightforward to reuse for the scratchpad-sized partial sort, by configuring the merge heap so each double-buffer in the AABB storage is the final one in its block, and no further buffers are fetched. Then the only additional hardware needed is logic to sort every buffer-sized sub-block prior to merging. This is easy to implement concurrently with data reading, by streaming the read data into a small number of buffer-sized presorters: our implementation consists of a hardware state machine performing an insertion sort at a rate of one compare-and-swap per second, plus one cycle of overhead for each inner loop of the sort. Multiplexers are inserted to bypass the insertion sorters in the multi-merge mode, and the hierarchy emitter in the partial sort mode.

5.6 Toplevel Build

Trees produced by HLBVH likely require post-processing for acceptable quality. The earliest idea proposed in this direction is HLBVH+SAH [42], where top levels of the BVH hierarchy, corresponding to high bits of the Morton code, are rearranged with a higher-quality algorithm. This toplevel build concept is of particular interest in a memory-constrained system, as the datasets are small enough to fit on-chip.

To evaluate toplevel builds, we add a configuration option to our simulated hardware to emit an array of high-level nodes, which can then be passed to a separate software or hardware builder. Two toplevel builders are evaluated in this work: a binned SAH build using the accelerator of Doyle et al. [13], and a software implementation of ATRBVH [12]. ATRBVH is itself a LBVH postprocessing step, but the usage in this article is novel in that we apply the algorithm only to the high-level nodes rather than processing the entire tree: It is then sufficiently lightweight to give real-time performance on a mobile CPU.

6. EVALUATION

This section describes first models used for comparison against the state-of-the-art BVH builder [13]. We then evaluate the performance, silicon area, and power characteristics of the proposed builder architecture in isolation. Finally, the builders are modeled as components of a larger rendering system.

6.1 Binned SAH Builder Model

The closest point of comparison for the proposed builder is the binned SAH architecture proposed by Doyle et al. [13]. The SAH builder is also of interest as a toplevel builder used to improve the tree quality of HLBVH trees output by the current work. For evaluation, it is interesting to examine the builder in more scenes than reported in Reference [13] and to consider its energy consumption.

First, memory traffic is modeled by instrumenting a software binned SAH builder to record build statistics. Traffic is caused by the BVH output and by *large sweeps* whose datasets are too long to fit in local primitive buffers. The architecture instance in Reference [13] has buffer capacity for 8,192 primitive AABBs. However, during each sweep, the hardware simultaneously produces input data for two child sweeps: In some cases, both children could individually fit in the buffers but are together too large. In this case, we assume only the smaller child to be a large sweep. As shown in Table II , this model comes close to replicating the memory traffic results in Reference [13]. A fixed size threshold of 4,096 slightly overestimates traffic, while a threshold of 8,192 underestimates it. Floating-point operations are counted based on Wald's algorithm [51] and then combined with memory traffic to obtain a lower-bound memory model.

Table II. Memory Traffic Model Validation

Scene	Mem. traffic (MB) Builder [13]	Estimated
Toasters	2	1
Fairy	25	25
Conference	120	125
Dragon	380	379

Traffic reported for the binned SAH builder [13] is compared against traffic predicted by the model.

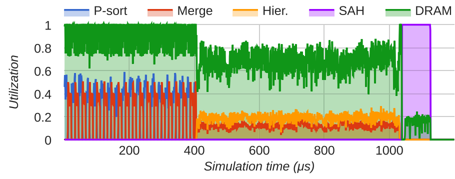
Finally, the runtime of a simplified binned SAH builder is modeled to give a lower bound for toplevel build performance. The simplified builder operates serially and has a single partitioning unit. It alternates between partitioning and binning a sweep at a rate of one input AABB per cycle, and SAH computation, which takes 32 cycles at the end of each sweep. The unit simplified in this way is substantially slower than the original, but as toplevel trees have only ca. 500–2,000 nodes in our test scenes, toplevel build has neglible effect on runtime.

6.2 Implementation and Power Analysis

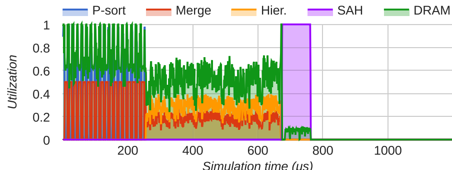
In the graphics hardware community, hardware complexity is often estimated by counting arithmetic units and memories in the design, but in this case we are especially interested in the energy of the proposed architecture, and whether it can reach a high clock frequency. Therefore, we wrote a prototype RTL description of the proposed architecture and synthesize it on a CMOS technology. All components in Figure 1 were implemented in SystemVerilog, and SRAM macros were used for the AABB storage.

The tree builder was synthesized on a 28nm FDSOI technology with Synopsys Design Compiler. The parameters of the builder were set at $M = 8,192$, $B = 16$, resulting in a unit with a 256KB scratchpad memory, which handles up to 2M triangles in one pass, reads data in 512B increments and performs a 256-way merge. We include eight partial sorters for scalability. To determine the buffer size B, we experimented with the DRAMPower model [10] and found that increasing consecutive access size is clearly beneficial at least up to 512B (16 AABBs). The target frequency is set at 1GHz, supply voltage at 1V, and operating temperature at 25°C. Clock gating and multi-threshold voltage optimizations are enabled.

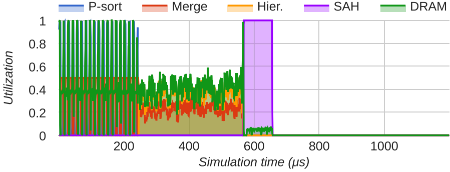
To evaluate performance, we run RTL simulations of the builder unit with various input scenes and memory interfaces. The external memory is modeled with the DRAM simulator Ramulator [26]. For the memory organization, we select 64-bit, one- and 2two-channel LPDDR3-1600 (*slow*, *medium*) that are the closest devices to state-of-the-art mobile device memory for which we are able to perform power analysis. In addition, we simulate a 64-bit, four-channel LPDDR3-1333 memory (*fast*) that gives a bandwidth close to Doyle [13] to facilitate a direct runtime comparison, but this interface is not representative of mobile systems. We subtract from all memory power figures a static power term computed from an idle memory transaction trace, corresponding to, for example, refresh power, to isolate the extra dynamic power added by the tree build. Ramulator is integrated to the RTL testbench through the



(a) LPDDR3-1600, 1x 64b channel (12.8 GB/s)



(b) LPDDR3-1600, 2x 64b channel (25.6 GB/s)



(c) LPDDR3-1333, 4x 64b channel (42.7 GB/s)

Fig. 6. Cycle-level simulation traces for the *sibenik* scene with three memory interface options. Utilizations of the multi-merge and hierarchy emitter components cap at 50%. Computation consists of partial sorts, a multi-merge phase, and an optional toplevel SAH build. With the slow bus, performance is limited by memory bandwidth. With the fast bus, memory latencies and compute pipeline throughputs become the bottleneck, particularly in the partial sort stage.

SystemVerilog Direct Programming Interface wrapper, such that memory requests from the simulated builder map into Ramulator transactions, and input data is fed into the builder when the corresponding read transaction completes. RTL simulation was run on 14 test scenes, and the resulting trees were verified in a software ray tracer.

We have also implemented a C++ architectural simulator that gives results within ca. 5% of the RTL simulation at a 20× faster runtime and allows easier observation of the build process. Figure 6 shows example simulation traces generated with the C++ simulator. The different execution states are visible: First, the unit alternates between partial sort reads that utilize the insertion sorters and writes that utilize the merge heap. Most of the execution time is spent on the multi-merge phase, which is clearly memory limited in the slower memory options. In the fast memory option, the throughput of the multi-merge hardware starts to limit performance. Finally, a toplevel SAH build is shown, which is more compute-intensive and uses little memory.

The build times, memory traffic and tree quality of the proposed builder are compared to related work. As a desktop benchmark, we compare against the high-quality ATRBVH builder by Domingues and Pedrini [12] with default settings, and their freely available implementation of Karras' HLBVH algorithm [22], set to use 32-bit Morton codes. The GPU builders are run on a computer with a GeForce GTX 1080 GPU and a Intel Core i7-3930K CPU, counting only kernel execution times. In tree builder hardware, we compare against the state-of-the-art binned SAH builder by Doyle et al. [13] and the k-d tree builder FastTree [34]. We use the performance figures from their article and generate memory traffic as described in the previous subsection. Memory traffic for GPU HLBVH was extracted with *nvprof*.

Tree quality is evaluated based on the SAH cost of produced trees [19]. The SAH cost C of a BVH can be computed as:

$$C = C_i \sum_{i=0}^{n_{\text{nodes}}} \frac{A(N_i)}{A(R)} + C_l \sum_{i=0}^{n_{\text{leafs}}} \frac{A(L_i)}{A(R)} + C_t \sum_{i=0}^{n_{\text{leafs}}} \frac{P_i A(L_i)}{A(R)},$$

where $A(N_i)$ and $A(L_i)$ are the surface areas of the given inner nodes and leafs, $A(R)$ the surface area of the scene AABB, P_i is the primitive count within a given leaf, C_i is the cost of traversing a node, C_l the cost of traversing a leaf, and C_t the cost of a primitive intersection test [23]. We use the SAH cost parameters $C_i = 1.2$, $C_l = 0$, $C_t = 1$ given for GPUs by Karras and Aila [23] to be comparable with previous work. SAH costs are normalized to a full SAH sweep [52]. Quality was not evaluated for FastTree.

Two toplevel builds are evaluated to recover quality, as discussed in Subsection 4.6. HLBVH+SAH is modeled with a cycle-level simulator, while for HLBVH+ATRBVH, we run a single-threaded C++ implementation on a NVidia Jetson TK1 board with a Tegra K1 SoC. The ATRBVH build runs two iterations over the toplevel nodes and uses a treelet size of 8.

For power analysis, we extract *switching activity information files* from the previous simulations, and perform power analysis with Synopsys Design Compiler. The constructed trees are loaded into a software ray tracer to verify correctness and compute tree quality. External memory power is determined by exporting DRAM command traces from Ramulator to DRAMPower [10]. We estimate the power and energy consumption of a 64b DRAM component by doubling the figures for a 32b component, as these could be combined into a 64b component.

6.3 Results

In Table III, the resulting build performance and tree quality is compared to related work. Even with the *slow* memory option, the present design is over 2× faster than the state of the art binned SAH unit, and with the *fast* memory option, 5× faster (6.3× including the Toasters scene, but the 1ms runtime reported in that scene is too imprecise for comparison). Compared to the state-of-the-art k-d tree builder by Liu et al. [34] with the same 12.8GB/s bandwidth, MergeTree is 4.7× faster. With the fast memory option, the proposed unit is within a factor of two of the desktop GPU HLBVH builder, which has 7.5× more memory bandwidth, and an orders of magnitude larger chip area and power envelope. ATRBVH gives a higher tree quality, but is, on average, 7.4× slower than HLBVH.

6.3.1 Area, Power, and Memory Traffic. The unit was successfully synthesized and meets timing constraints at 1GHz. The cell area and power breakdown of the synthesized unit is shown in Table IV. Table V shows an area comparison to related work. The proposed unit has ca. 2.5× less area than a binned SAH builder [13].

The results of power analysis are shown in Table VII. The main result is that over 90% of total power consumption in the design comes from the DRAM interface. There are some straightforward optimizations to reduce the on-chip power: For example, a multi-bank scratchpad could be used in place of the current expensive

Table III. Build Performance and Quality Comparison

	BW (GB/s)	Toasters (11K) Time (ms)	SAH	Bunny (70K) Time (ms)	SAH	Cloth (92K) Time (ms)	SAH	Fairy (174K) Time (ms)	SAH
GTX 1080 HLBVH [12]	320	0.27	166%	0.45	103%	0.52	123%	0.82	129%
GTX 1080 ATRBVH [12]	320	1.44	76%	2.67	89%	3.35	92%	6.03	87%
HW binned SAH [13]	44	1	103%	-	104%	3	103%	-	102%
HW k-d tree [34]	12.8	-	-	5.1	-	-	-	10.8	-
HW HLBVH (proposed)	12.8	0.16	160%	1.11	122%	1.50	122%	2.14	118%
—"—	25.6	0.10	"	0.66	"	0.91	"	1.46	"
—"—	42.7	0.08	"	0.49	"	0.66	"	1.08	"
HW Binned SAH topl.	12.8	0.03	112%	0.11	113%	0.05	114%	0.05	104%
TK1 ATRBVH topl.	14.9	4.60	102%	28.92	116%	9.31	113%	6.35	109%

	Mem. BW	Crytek (262K) Time (ms)	SAH	Conference (283K) Time (ms)	SAH	Sportscar (301K) Time (ms)	SAH	Italian (375K) Time (ms)	SAH
GTX 1080 HLBVH [12]	320	1.10	151%	1.25	150%	1.22	184%	1.45	214%
GTX 1080 ATRBVH [12]	320	8.11	87%	10.01	96%	9.45	89%	11.69	85%
HW binned SAH [13]	44	-	107%	11	103%	-	120%	-	105%
HW k-d tree [34]	12.8	-	-	17.2	-	-	-	-	-
HW HLBVH (proposed)	12.8	3.60	142%	2.83	154%	4.17	142%	4.80	208%
—"—	25.6	2.40	"	2.05	"	2.77	"	3.26	"
—"—	42.7	1.71	"	1.88	"	1.98	"	2.35	"
HW Binned SAH topl.	12.8	0.06	109%	0.02	106%	0.08	121%	0.04	135%
TK1 ATRBVH topl.	14.9	10.05	99%	4.33	89%	15.20	116%	8.62	117%

	Mem. BW	Babylonian (500K) Time (ms)	SAH	Kitchen (761K) Time (ms)	SAH	Dragon (871K) Time (ms)	SAH	Buddha (1087K) Time (ms)	SAH
GTX 1080 HLBVH [12]	320	1.80	208%	2.49	155%	2.99	136%	3.66	133%
GTX 1080 ATRBVH [12]	320	14.69	91%	19.80	91%	23.22	90%	29.55	84%
HW binned SAH [13]	44	-	104%	-	101%	30.00	102%	-	102%
HW k-d tree [34]	12.8	-	-	-	-	52.5	-	65.5	-
HW HLBVH (proposed)	12.8	6.52	202%	8.61	149%	13.08	135%	15.60	132%
—"—	25.6	4.43	"	6.07	"	8.55	"	10.32	"
—"—	42.7	3.21	"	4.46	"	6.04	"	7.32	"
HW binned SAH topl.	12.8	0.04	147%	0.02	113%	0.08	120%	0.06	120%
TK1 ATRBVH topl.	14.9	7.54	128%	3.98	94%	17.60	123%	12.84	122%

	Mem. BW	Livingroom (1459K) Time (ms)	SAH	Lion (1604K) Time (ms)	SAH	Geom. Mean Norm. time	SAH
GTX 1080 HLBVH [12]	320	4.41	160%	5.07	144%	0.68	153%
GTX 1080 ATRBVH [12]	320	39.84	82%	42.37	88%	5.14	89%
HW binned SAH [13]	44	-	101%	-	103%	6.40	104%
HW k-d tree [34]	12.8	-	-	-	-	9.39	-
HW HLBVH (proposed)	12.8	16.76	144%	24.05	139%	2.02	148%
—"—	25.6	11.76	"	15.76	"	1.33	"
—"—	42.7	8.56	"	11.12	"	1.00	"
HW binned SAH topl.	12.8	0.06	116%	0.28	128%	0.03	118%
TK1 ATRBVH topl.	14.9	9.69	109%	45.75	119%	4.94	111%

-

SAH costs are relative to a full SAH sweep. Average build time is normalized to GTX 1080 HLBVH. BW denotes system memory bandwidth. The proposed builder (HW HLBVH) is evaluated with three DRAM configurations. Toplevel Builds (HW Binned SAH, TK1 ATRBVH) show build time *in Addition to HW HLBVH*.

Table IV. Area and Power Breakdown of Synthesized Design Power results from the Lion scene, fast memory model

	Area (mm^2)	Power (mW)
Insertion sorters	0.24	25.8
Multi-merge unit	1.14	17.2
Hierarchy emitter	0.03	2.2
FIFOs and muxes	0.99	16.0
Total	2.41	61.2

Table V. Hardware Comparison with Quadratic Process Scaling

Architecture	Area (mm^2)	Node (nm)	Area @28nm (mm^2)	Mem. BW (GB/s)
Geforce GTX 1080	314	16	962	320
HW binned SAH [13]	31.9	65	5.9	44
HW k-d tree [34]	1.4	28	1.4	43
MergeTree	2.4	28	2.4	43

Table VI. Memory Traffic Comparison (MB)

Scene	Proposed	HW Binned SAH	GPU HLBVH
Toasters	1.7	1.1 (0.7x)	1.7 (1.0x)
Bunny	10.6	18 (1.6x)	27 (2.5x)
Cloth	14.2	25 (1.7x)	36 (2.5x)
Fairy	19.4	70 (3.6x)	74 (3.8x)
Crytek	32.9	116 (3.5x)	115 (3.5x)
Conference	28.9	125 (4.3x)	125 (4.3x)
Sportscar	38.6	110 (2.9x)	135 (3.5x)
Italian	43.9	145 (3.3x)	169 (3.9x)
Babylonian	60.2	219 (3.6x)	230 (3.8x)
Kitchen	77.1	436 (5.6x)	352 (4.6x)
Dragon	124.8	379 (3.0x)	413 (3.3x)
Buddha	147.9	493 (3.3x)	520 (3.5x)
Livingroom	150.4	833 (5.5x)	685 (4.6x)
Lion	230.1	800 (3.5x)	766 (3.3x)
Geom. mean	-	(3.0x)	(3.3x)

dual-port SRAM. However, since the power consumed by the hardware unit itself is negligible compared to DRAM, the improvements from optimization would also be marginal.

Table VI compares our memory traffic to related work. The proposed builder generates 3.0× less traffic than hardware binned SAH, and 3.3× less than a GPU build—the radix sort stage of the GPU build alone generates roughly as much traffic as our complete build. Our builder also achieves very high bus utilizations of 72%, 54%, and 44% of the memory bandwidth on the slow, medium and fast interface options, respectively.

The above results show that the energy consumption and build speed of MergeTree are largely determined by the amount of memory traffic generated. A straightforward conversion of HLBVH to hardware, without the proposed memory traffic optimizations, would likely have a ca. 3× higher energy consumption and runtime, almost as high as the binned SAH builder [13].

6.3.2 Tree Quality. MergeTree builds slightly higher-quality trees than the GPU HLBVH builder, since we generate large leafs for primitives with identical Morton codes, as in Reference [42], while the GPU builder organizes these primitive ranges with an arbitrary subtree and emits one leaf per triangle. Nevertheless, our plain HLBVH trees have low quality, on average 148%, compared to 104% for Doyle et al. [13].

Table VII. Power Analysis Results, Average of 14 Scenes

Max. BW (GB/s)	12.8	25.6	42.7
Mem. traffic (GB/s)	9.2	13.9	18.9
Logic power (mW)	58.1	61.8	62.7
DRAM power (mW)	507.2	814.8	1,112.4
Total power (mW)	565.3	876.6	1,175.2

The evaluated toplevel builds give significant tree quality improvements: HLBVH+SAH to an average of 118% and HLBVH+ATRBVH to 111%, only ca. 5% worse than a binned SAH build. The hardware binned SAH builder has an insignificant runtime compared to the LBVH but consumes extra chip area. Our naíve single-threaded software implementation of toplevel ATRBVH is already fast enough for real-time construction on a mobile SoC and could run all scenes at 30FPS except for Lion, which has demanding geometry.

6.4 System Level Comparison

From the previous results, it is apparent that MergeTree gives a similar tradeoff as desktop GPU HLBVH: It is very fast and energy efficient compared to prior work, at the cost of reduced tree quality, which can be mostly recovered with postprocessing toplevel builds. We can conjecture that a binned SAH builder is advantageous in small scenes where the build effort is minuscule relative to rendering, and the proposed builder becomes advantageous in larger scenes. The exact tradeoff depends on the particular scene and visual effects being displayed. This subsection further quantifies the system-level effects of builder selection by modeling a larger system that includes rendering hardware. The model focuses on system energy consumption per frame, as it is a main figure of interest in mobile systems and simplifies modeling. We start out from the premise that the BVH tree for the complete scene is rebuilt for each frame, and a viewpoint is then rendered at a $1,280 \times 720$ resolution. The scenes and viewpoints used are shown in Table III. Benchmarks are run for primary ray rendering, as well as diffuse lighting with one sample per pixel, limited to three bounces. The latter is representative of incoherent secondary rays.

The main components of the present model are a fixed-function rendering accelerator, combined with MergeTree, binned SAH and ATRBVH hardware builders. Moreover, toplevel build combinations of HLBVH+SAH and HLBVH+ATRBVH are tested that combine two hardware builders. For MergeTree, we use accurate energy figures based on post-synthesis power analysis and DRAMPower. For the binned SAH builder, we estimate memory traffic and FPU operation counts as described in Subsection 5.1 and then obtain a lower-bound energy model by assuming fully utilized FPUs and long, consecutive burst accesses to DRAM. For the ATRBVH builder, memory accesses and FPU operations are likewise counted from program code. No high-performance hardware architecture has been published for ATRBVH, but given the input size for toplevel builds, even a serial hardware unit performing one FPU operation per cycle is sufficient to process all test scenes at over 100fps. Finally, the rendering accelerator is modeled after the traversal and intersection unit of SGRT [32] and simulated at cycle level as described in Reference [50]. Some common assumptions are used when modeling the hardware units. The units reside on a mobile SoC that is fabricated with a 28nm process technology and

equipped with the 25.6GB/s memory interface described earlier. As with MergeTree, off-chip memory accesses are modeled with Ramulator and DRAMPower. Caches and SRAMs are parametrized with CACTI 6.5 [38]. Floating-point unit energy is based on the figures of Galal et al. [15], with linear process scaling, as in the GPUSimPow simulator [35]. Reciprocal calculation is estimated to take as much energy as 3 FLOPs, as in Reference [33]. All units beside MergeTree operate at 500MHz.

6.4.1 Rendering Hardware Model. The modeled accelerator architecture is shown in Figure 8: It consists of separate fixed-function pipelines for tree traversal and primitive intersection. Scenes are rendered with a software ray tracer, from which a traversal trace is extracted and fed to a cycle-level hardware simulator, which traces utilizations for all components in Figure 8. The power consumption of each component is determined by multiplying a dynamic power term with the utilization and adding a static power term. For the fixed-function pipelines, the energy consumption of floating-point adds, multiplications and reciprocals is counted. The architecture and simulation flow is described in more detail in Reference [50] and is unchanged from that work, except the SRAM and FPU models are updated to a 28nm process.

In addition to tree construction and traversal, shading is the third main component in the rendering process. Shading has a very wide range of complexity: We experimented with adding the energy cost of minimal shading and pixel output to the model, that is, Phong shading with a directional light, but this had minimal effect on the total power. On the other hand, sufficiently complex shading may dominate the rendering process [29]. Mobile ray tracing would likely opt for inexpensive shaders at first. As shading cost is independent of tree quality, we omit it from the model.

6.4.2 Results. The system-level energy results are shown scene by scene in Figure 7 and summarized in Table VIII. It is visible that the energy cost of tree construction scales asympotically faster with scene size than traversal and, with binned SAH, dominates the energy profile in large scenes. Though the binned SAH builder performs significant floating-point computation, most of its energy consumption is also due to DRAM accesses. MergeTree uses on average ca. 3.2× less energy. In primary ray tracing, the build energy savings are sufficient to make HLBVH preferred in all scenes except Toasters. In large scenes, tree construction dominates the system energy, and HLBVH gives significant savings.

With incoherent secondary rays, the energy footprint of ray tracing grows significantly and is dominated by memory traffic. As such, tree quality has a larger effect on system energy. Moreover, the tracing energy penalty of the proposed builder is larger than predicted by SAH cost. Toplevel builds reduce system energy but less than predicted by SAH. Regardless, in large (>500K triangle) scenes the cost of tree construction is significant enough that the proposed builder consistently reduces system energy compared to binned SAH.

For comparison with mobile power budgets and GPUs, the energy results in Figure 7 are presented as power at a fixed 30FPS frame rate. Diffuse, animated ray tracing in our model with HLBVH+ATRBVH dissipates between 143..2077mW of power, and primary ray tracing between 72..791mW. For a point of comparison, in the benchmarks of Pathania et al. [43], recent mobile games on an Odroid-XU+E platform dissipate ca. 2..3W, of which ca. 0.8..1.8W is used in the mobile GPU. These results suggest that MergeTree allows the ray tracing of large (>500K triangle) dynamic scenes in a mobile power envelope. However, in the most demanding scenes there is only limited margin for complex shading. A binned SAH builder would already use most of the mobile power budget for these scenes.

Table VIII. System Energy with Different Builders, Main Results. Energy normalized to binned SAH [Doyle et al. 2013], averaged over 14 scenes

Energy	Binned SAH	HLBVH	HLBVH +SAH	HLBVH +ATRBVH
Primary rays, all scenes				
Build	100%	**32%**	33%	33%
Trace	**100%**	143%	114%	112%
Total	100%	71%	59%	**58%**
Primary rays, large scenes				
Build	100%	**22%**	22%	22%
Trace	**100%**	148%	122%	122%
Total	100%	41%	37%	**37%**
Diffuse rays, all scenes				
Build	100%	**32%**	33%	33%
Trace	**100%**	163%	134%	128%
Total	100%	110%	91%	**87%**
Diffuse rays, large scenes				
Build	100%	**22%**	22%	22%
Trace	**100%**	165%	141%	137%
Total	100%	79%	68%	**65%**

7. LIMITATIONS AND FUTURE WORK

One difficulty in the proposed design is handling scenes of over $\frac{M^2}{2B}$ primitives (2 million triangles with the evaluation setup), as they require more than one multi-merge pass. It is simple to add control logic for multiple passes, but the use of AABBs as sorting elements is then suboptimal. Another possibility is to enlarge the scratchpad: Doubling the memory size M quadruples the model size that can be processed in one pass. In our experience, at least a 512KB scratchpad memory can run at 1GHz; this would be sufficient for scenes of 8M triangles. It would also be interesting to evaluate, as a replacement for double buffering, the other well-known multi-merge sort read scheduling technique of *forecasting*, used by, for example, Barve et al. [7]. Forecasting introduces a second heap that stores the final elements of each buffer and uses it to fetch replacement buffers in an optimal order. Forecasting would, again, double the size of scene that can be sorted in a single pass with a given scratchpad size.

Recent trends in ray-tracing accelerators are toward techniques that reduce the cost of ray traversal while complicating tree construction, for example, compressed BVHs [25] and treelet scheduling [3]. In future architectures incorporating these features, the cost of tree construction will be emphasized even more than in straightforward single-precision traversal as evaluated in this article. We are extending MergeTree to generate compressed trees.

This article focused on mobile ray tracing, but the design also has interesting applications in, for example, collision detection as in Reference [11], and with minor modifications, construction of point set k-d trees [22] and data sorting.

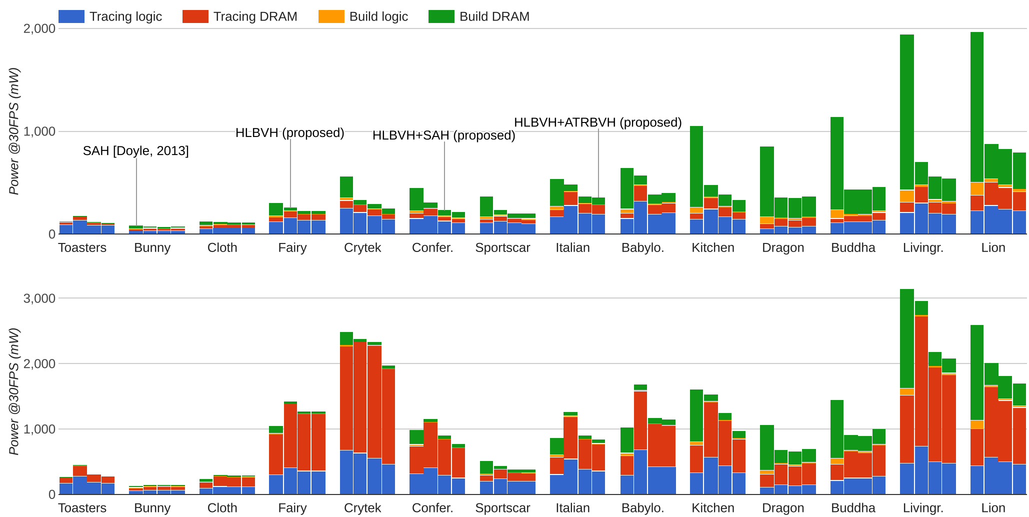


Fig. 7. System energy with only primary rays (top) and diffuse secondary rays (bottom), expressed as power at 30FPS for comparison to mobile GPUs. Four alternatives are modeled: HLBVH only, binned SAH only, and HLBVH with binned SAH and ATRBVH toplevel builds. HLBVH build energy is from RTL simulation, while tracing and binned SAH energy are based on higher-level models. The proposed HLBVH builder reduces build energy at the cost of worse tree quality, which increases tracing energy. Toplevel builds remove much of the quality penalty. Tracing complexity is weakly related to scene size, while build energy grows, and becomes dominant in large scenes. Averaged results are shown in Table VIII.

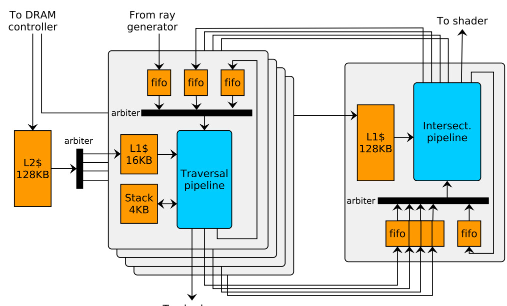


Fig. 8. Ray-tracing accelerator simulated for system level energy modeling. The accelerator is modeled after the traversal and intersection unit in SGRT [32], with a two-AABB node layout [31]. All shown components are simulated at cycle level.

8. CONCLUSION

In this article, we proposed MergeTree, the first hardware accelerator architecture for the HLBVH algorithm, which forms the basis for the highest-performance GPU tree construction algorithms. Novel techniques were proposed to adapt HLBVH into a streaming, serial hardware form, which is suitable for mobile systems with limited power budgets, due to reduced memory traffic. Our results show significant improvements over previous state of the art [13] in terms of build performance, silicon area, memory traffic and energy consumption, at the cost of reduced tree quality, which can be mitigated with inexpensive toplevel builds.

The proposed architecture substantially increases the size of animated scenes that could be rendered by a mobile ray-tracing accelerator in real time, and also has applications outside ray tracing. System-level modeling showed that the cost of tree construction begins to rival the cost of real-time rendering in large scenes. MergeTree gives significant system energy savings in these scenes.

ACKNOWLEDGMENTS

This work was financially supported by the TUT graduate school and the Academy of Finland (decision #297548, PLC). The 3D models used are courtesy of Ingo Wald (Fairy), Andrew Kensler (Toasters), Yasutoshi Mori (Sportscar), Frank Meinl (Crytek Sponza), Jonathan G. (Italian, Babylonian), Anat Grynberg and Greg Ward (Conference), Naga Govindaraju, Ilknur Kabul and Stephane Redon (Cloth), the Stanford Computer Graphics Laboratory (Bunny, Dragon), and the SceneNet library [21] (Livingroom, Kitchen). Crytek Sponza and Dragon have modifications courtesy of Morgan McGuire [37].

REFERENCES

[1] Attila T. Áfra and László Szirmay-Kalos. 2014. Stackless multi-BVH Traversal for CPU, MIC and GPU ray tracing. *Comput. Graph. Forum* 33, 1 (2014), 129–140.

[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[3] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of High Performance Graphics*. 113–122.

[4] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics*. 145–149.

[5] Tomas Akenine-Möller and Jacob Strom. 2008. Graphics processing units for handhelds. *Proc. IEEE* 96, 5 (2008), 779–789.

[6] Ciprian Apetrei. 2014. Fast and simple agglomerative LBVH construction. In *Proceedings of the Computer Graphics and Visual Computing Conference (CGVC'14)*.

[7] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. 1997. Simple randomized mergesort on parallel disks. *Parallel Comput.* 23, 4 (1997), 601–631.

[8] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2. 538–547.

[9] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 151–160.

[10] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. Retrieved February 30, 2017 from http://www.drampower.info.

[11] Erwin Coumans. 2017. Bullet physics library. Retrieved March 6, 2017 from http://www.bulletphysics.org.

[12] Leonardo R. Domingues and Helio Pedrini. 2015. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of High Performance Graphics*. 13–20.

[13] Michael Doyle, Colin Fowler, and Michael Manzke. 2013. A hardware unit for fast SAH-optimized BVH construction. *ACM Trans. Graph.* 32, 4 (2013), 139:1–10.

[14] Michael Doyle, Ciaran Tuohy, and Michael Manzke. 2017. Evaluation of a BVH construction accelerator architecture for high-quality visualization. *IEEE Trans. Multi-Scale Comput. Syst*. Early access. Retrieved from http://ieeexplore.ieee.org/abstract/document/7903616/.

[15] Sameh Galal and Mark Horowitz. 2011. Energy-efficient floating-point unit design. *IEEE Trans. on Comput.* 60, 7 (2011), 913–922.

[16] Per Ganestam and Michael Doggett. 2016. SAH guided spatial split partitioning for fast BVH construction. *Comput. Graph. Forum* 35, 2 (2016), 285–293.

[17] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011a. Simpler and faster HLBVH with work queues. In *Proceedings of High Performance Graphics*. 59–64.

[18] Kirill Garanzha, Simon Premože, Alexander Bely, and Vladimir Galaktionov. 2011b. Grid-based SAH BVH construction on a GPU. *Vis. Comput.* 27, 6–8 (2011), 697–706.

[19] Jeffrey Goldsmith and John Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.* 7, 5 (1987), 14–20.

[20] Aggelos Ioannou and Manolis G. H. Katevenis. 2007. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.* 15, 2 (2007), 450–461.

[21] Ilan Kadar and Ohad Ben-Shahar. 2013. SceneNet: A perceptual ontology database for scene understanding. *J. Vis.* 13, 9 (2013), 1310–1310.

[22] Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of High Performance Graphics*. 33–37.

[23] Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of High Performance Graphics*. 89–99.

[24] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17.

[25] Sean Keely. 2014. Reduced precision hardware for ray tracing. In *Proceedings of High Performance Graphics*. 29–40.

[26] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Comput. Arch. Lett.* PP, 99 (2015), 1–1.

[27] Dirk Koch and Jim Torresen. 2011. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 45–54.

[28] Daniel Kopta, Konstantin Shkurko, J. Spjut, Erik Brunvand, and Al Davis. 2015. Memory considerations for low energy ray tracing. *Comput. Graph. Forum* 34, 1 (2015), 47–59.

[29] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of High Performance Graphics*. 137–143.

[30] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. *Comput. Graph. Forum* 28, 2 (2009), 375–384.

[31] Jaedon Lee, Won-Jong Lee, Youngsam Shin, Seokjoong Hwang, Soojung Ryu, and Jeongwook Kim. 2014. Two-AABB traversal for mobile real-time ray tracing. In *Proceedings of the SIGGRAPH Asia Symposium on Mobile Graphics and Interactive Applications* 14.

[32] 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 High Performance Graphics*. 109–119.

[33] Wei Liu and Alberto Nannarelli. 2012. Power efficient division and square root unit. *IEEE Trans. Comput.* 61, 8 (2012), 1059–1070.

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

[35] Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. 2013. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis and System Software*. 97–106.

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

[37] Morgan McGuire. 2011. Computer graphics archive. Retrieved Feb 30, 2017 from http://graphics.cs.williams.edu/data/meshes.xml.

[38] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report. HP Laboratories. 22–31 pages.

[39] J. Nah, H. Kwon, D. Kim, C. Jeong, J. Park, T. Han, D. Manocha, and W. Park. 2014. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph.* 33, 5 (2014), 162:1–15.

[40] Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han. 2015. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Trans. Vis. Comput. Graph.* 21, 3 (2015), 389–401.

[41] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I engine: Traversal and intersection engine for hardware accelerated ray tracing. *ACM Trans. Graph.* 30, 6 (2011), 160.

[42] Jacopo Pantaleoni and David Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of High Performance Graphics*. 87–95.

[43] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. 2015. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Proceedings of the Design Automation Conference*. 201.

[44] PowerVR. 2015. PowerVR Ray Tracing. Retrieved Feb 30, 2017 from https://imgtec.com/powervr/ray-tracing/.

[45] Maxim Shevtsov, Alexei Soupikov, Alexander Kapustin, and Nizhniy Novorod. 2007. Ray-triangle intersection algorithm for modern CPU architectures. In *Proceedings of GraphiCon*, Vol. 2007. 33–39.

[46] Hojun Shim, Nachyuck Chang, and Massoud Pedram. 2004. A compressed frame buffer to reduce display power consumption in mobile systems. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 819–824.

[47] Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A multicore hardware architecture for real-time ray tracing. *Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 12 (2009), 1802–1815.

[48] Joseph Spjut, Daniel Kopta, Erik Brunvand, and Al Davis. 2012. A mobile accelerator architecture for ray tracing. In *Proceedings of the Workshop on SoCs, Heterogeneous Architectures and Workloads*.

[49] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2015. MergeTree: A HLBVH constructor for mobile systems. In *Proceedings of SIGGRAPH Asia, Technical Briefs*. 12.

[50] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, and Jarmo Takala. 2016. Multi bounding volume hierarchies for ray tracing pipelines. In *Proceedings of SIGGRAPH Asia, Technical Briefs*. 8.

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

[52] Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.* 26, 1 (2007), 6.

[53] Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.* 24, 3 (2005), 434–444.

Received March 2017; accepted June 2017

[P3] Publication 3

[P3] Viitanen T., Koskela M., Jääskeläinen P., Immonen K., Takala J.:
Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies.
Computer Graphics Forum 36, 4 (2017), 167−178.

DOI: 10.1111/cgf.13233

Eurographics Symposium on Rendering 2017
P. Sander and M. Zwicker
(Guest Editors)

Volume 36 (2017), Number 4

Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies

T. Viitanen†, M. Koskela, P. Jääskeläinen, K. Immonen and J. Takala

Tampere University of Technology, Finland

Abstract

There is recent interest in GPU architectures designed to accelerate ray tracing, especially on mobile systems with limited memory bandwidth. A promising recent approach is to store and traverse Bounding Volume Hierarchies (BVHs), used to accelerate ray tracing, in low arithmetic precision. However, so far there is no research on refitting or construction of such compressed BVHs, which is necessary for any scenes with dynamic content. We find that in a hardware-accelerated tree update, significant memory traffic and runtime savings are available from streaming, bottom-up compression. Novel algorithmic techniques of modulo encoding and treelet-based compression are proposed to reduce backtracking inherent in bottom-up compression. Together, these techniques reduce backtracking to a small fraction. Compared to a separate top-down compression pass, streaming bottom-up compression with the proposed optimizations saves on average 42% of memory accesses for LBVH construction and 56% for refitting of compressed BVHs, over 16 test scenes. In architectural simulation, the proposed streaming compression reduces LBVH runtime by 20% compared to a single-precision build, and 41% compared to a single-precision build followed by top-down compression. Since memory traffic dominates the energy cost of refitting and LBVH construction, energy consumption is expected to fall by a similar fraction.

CCS Concepts
•***Computing methodologies*** → *Ray tracing; Graphics processors;*

1. Introduction

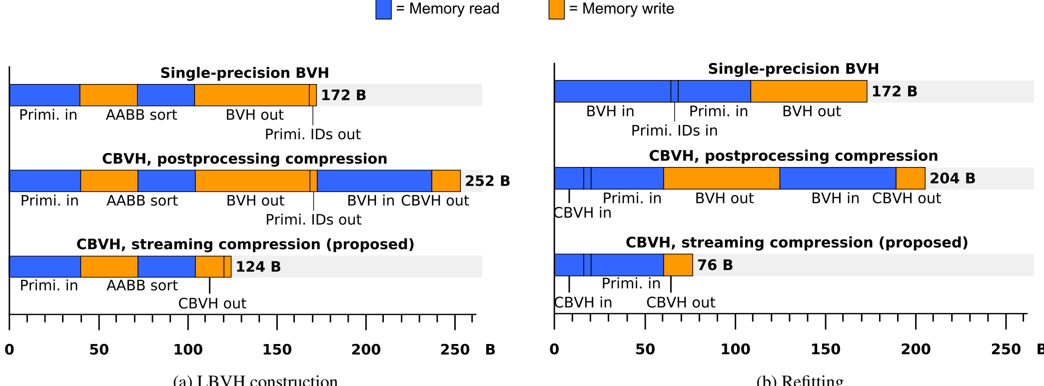
In the last decade, ray tracing GPU architectures have been the subject of intensive research and industrial interest. Ray tracing accelerators have been proposed based on fixed-function pipelines [LSL*13, NKK*14, WSS05, LV16], programmable MIMD processors [SKKB09, SKBD12], and augmenting conventional GPUs [Kee14]. There has been a special focus on architectures targeting mobile systems [LSL*13, NKK*14, SKBD12, Pow15], where they might, e.g., enable photorealistic augmented reality applications [LSL*13]. High-performance ray tracing is based on indexing the scene geometry in an acceleration datastructure, typically a *Bounding Volume Hierarchy* (BVH), which speeds up ray-scene collision queries. Recently, a potential breakthrough in ray tracing GPU architecture is to store and traverse BVHs at low arithmetic precision, e.g., 5-6 bits per coordinate. We refer to these structures as *Compressed BVHs* (CBVH). Keely [Kee14] estimates that the use of CBVHs – combined with compact leaf storage and treelet scheduling – reduces the arithmetic energy cost of ray traversal by 23x, and memory traffic by 6–22x.

However, there is no research yet on updating CBVHs in real time to match animated scenes. Many rendering applications include dynamic scene content, and a key advantage of plain BVHs has been the ability to rapidly construct new BVHs, or to *refit* an existing BVH to match deformed geometry [WBB08]. It is interesting whether this advantage is preserved in CBVH. Fast uncompressed BVH update algorithms have been studied extensively for GPUs [LGS*09, KIS*12], and hardware accelerators have been proposed [DFM13, VKJ*15, NKP*15]. The reduced cost of rendering in CBVHs will, through Amdahl's law, increase the relative share of tree updates, and make their performance more critical to the system. Meanwhile, compression introduces new complications to tree updates.

Tree updates are highly memory-intensive, and especially in the context of mobile systems and custom hardware pipelines, the amount of external memory traffic they generate is a major factor in their performance and energy efficiency [DFM13, VKJ*15]. Figure 1 shows the basic motivation and goal of this paper. A straightforward way to update a CBVH is to first produce a full-precision BVH tree and then subsequently compress it. We would like to, instead, directly output a CBVH in a *streaming* manner, saving up to ca. 50% of memory accesses for construction and 64% for refitting.

† e-mail: timo.2.viitanen@tut.fi

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.



(a) LBVH construction (b) Refitting

Figure 1: *External memory traffic of different hardware tree update strategies, bytes per input primitive; assuming LBVH unit similar to [VKJ*15] and trees with one primitive per leaf. Updating or refitting to single-precision BVH and postprocessing into CBVH has a large overhead compared to direct CBVH output. (Up to ca. 2x more traffic for LBVH and 2.6x for refit.)*

Streaming compression is straightforward with a top-down algorithm, but the fastest approaches to tree update, *refitting* [WBS07] and optimized implementations of *Linear BVH* (LBVH) [Ape14], output BVHs in bottom-up order. Bottom-up compression, in turn, is nontrivial since each CBVH node is encoded relative to its parent. Consequently, in order to emit a node, we need to make assumptions about its parent, and backtrack to repair the hierarchy when said assumptions are falsified. The memory traffic from backtracking can eclipse the traffic saved by streaming compression.

In this article, we investigate algorithmic techniques for bottom-up streaming compression of CBVHs, intended for a possible hardware implementation. The main contributions of this paper are as follows:

- A streaming, bottom-up CBVH compression algorithm based on context estimation and backtracking.
- A novel *modulo encoding* for CBVH coordinates which reduces the backtracking overhead by ca. $\frac{2}{3}$.
- A treelet-based bottom-up compression algorithm, which eliminates roughly a further $\frac{2}{3}$ of backtracking per level of treelet depth used.

Together, these techniques reduce backtracking to a small fraction, so that streaming compression gives close to ideal traffic savings.

The proposed algorithms could be used to implement a streaming compression hardware unit which would operate on the outputs of a hardware tree builder or refitter, reducing memory traffic, and consequently energy and runtime. The significance of the savings on a complete ray tracing system depend heavily on the scene and used visual effects. The best case is a scene with simple shading and complex animated geometry. In our preliminary simulations, the cost of LBVH construction is roughly equal to the cost of single-precision primary ray tracing at 1M animated triangles, so in this optimistic case, halving the build cost could give energy and runtime savings as high as $\frac{1}{4}$. Large savings are easier to reach with CBVHs as they are cheaper to traverse.

This paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the baseline CBVH encoding. Section 4 discusses the proposed approach of bottom-up streaming compressions, and proposes optimizations to reduce backtracking memory traffic inherent in the approach. Section 5 gives an evaluation of the proposed techniques, and Section 6 concludes the paper.

2. Related work

BVH compression with quantized coordinates has been investigated in software by Mahovsky [MW06] and later Segovia [SE10] for the purpose of out-of-core ray tracing of very large static scenes. Keely [Kee14] proposed a hardware architecture based on augmenting a conventional GPU with CBVH traversal hardware. A *traversal point update* method was proposed to traverse CBVHs with low-precision arithmetic computations, allowing traversal with compact hardware accelerators. Vaidyanathan et al. [VAMS16] approached CBVHs from a more formal framework and gave an alternate, provably watertight traversal algorithm, while reusing AABB coordinates from parent nodes as in [FD09] to further shrink the memory footprint and arithmetic cost of CBVHs.

The quality of BVH trees is typically measured with the *Surface Area Heuristic* (SAH) [GS87]. The classical build methods of SAH sweep and binned SAH sweep [Wal07] recursively partition the scene primitives guided by SAH values of candidate splits. Most tree construction methods aiming for fast build speed are based on the *Linear BVH* (LBVH) approach of Lauterbach et al. [LGS*09], which has been optimized by several authors [PL10, GPM11, Kar12, Ape14]. LBVH is a fast, low-quality build algorithm, and state-of-the-art builders further process the resulting tree to improve quality; e.g., the *Agglomerative Treelet Restructuring BVH* algorithm (ATRBVH) algorithm [DP15] rearranges an LBVH tree to give better tree quality than SAH sweep at a fraction of the runtime.

Another broad approach to tree update is to refit an existing

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

tree to new geometry [WBS07]. This is a faster operation than full rebuild, but restricted to animations that conserve mesh topology. Tree quality tends to deteriorate over many refits, so it is often periodically refreshed with an asynchronous high-quality rebuild [IWP07], or maintained with tree rotations during the refit [KIS*12].

Hardware accelerators have been proposed to update trees especially on mobile platforms, where energy and memory bandwidth constraints prevent the use of GPGPU algorithms. Nah et al. [NKP*15] propose a scheme similar to [IWP07] based on refitting and asynchronous rebuilds, however, refitting is accelerated with hardware *Geometry and Tree Update* (GTU) units. Doyle et al. [DFM13] proposed a hardware BVH builder based on the binned SAH sweep algorithm, which optimizes memory traffic by performing stages of the algorithm in a pipelined manner. Viitanen et al. [VKJ*15] have proposed an LBVH builder which gives a similar tradeoff as desktop LBVH compared to [DFM13]: the build is much faster, at the cost of reduced tree quality, which needs to be recovered with postprocessing.

In this paper, we describe how to augment LBVH and refitting hardware such as [VKJ*15, NKP*15] to emit CBVHs in a streaming manner. To our best knowledge, this is the first study on CBVH construction and refitting. The present work is focused on hardware implementation, but may also be applicable to GPGPU tree updates.

3. Background

We start from a formal description of quantized trees similar to the one given by Vaidyanathan et al. [VAMS16]. In a full precision BVH, each node is represented by an *Axis-Aligned Bounding Box* (AABB) consisting of a lower bound p and a upper bound q, where $(p,q) \in \mathbb{R}^3$. We write the components of vectors as, e.g., p_x, p_y, p_z. In CBVH, the bounds are quantized to a low resolution grid aligned with the parent AABB, represented as low-precision (e.g. 5–6 bit) integers, and decompressed during traversal. This results in a lossy compression where the AABB's memory footprint is sharply reduced, at the cost of some extra node tests due to enlarged bounds. If the quantized grids are zero-aligned, the local grid coordinates (r_i, s_i) for each axis i can be computed as

$$r_i = \left\lfloor \frac{p_i - u_i^{parent}}{2^{e_i^{\text{parent}}}} \right\rfloor; \quad s_i = \left\lfloor \frac{q_i - u_i^{parent}}{2^{e_i^{\text{parent}}}} \right\rfloor + 1, \qquad (1)$$

where e is the local grid scale exponent, and u^{parent} is the quantized parent lower bound. The per-axis grid scale exponents e_i of a node are chosen to be the minimum where the node fits in 2^N grid intervals, so that its r, s can be expressed in N bits, i.e.,

$$e_i = \arg\min_k \left((v_i - u_i)/2^k \leq 2^N \right). \qquad (2)$$

Note that s_i in Equation 1 takes on values between $1 \ldots 2^N$, which may overflow if stored in N bits. This can be avoided by storing $s_i - 1$.

When traversing the tree, the decompressed bounds (u, v) can be computed as

$$u_i = u_i^{\text{parent}} + r_i 2^{e_i^{\text{parent}}}; \qquad v_i = u_i^{\text{parent}} + s_i 2^{e_i^{\text{parent}}}. \qquad (3)$$

The original bounds (p,q) are enclosed in the decompressed bounds, such that $u \leq p \leq q \leq v$. Note that in order to encode or traverse a CBVH node with Equations 1 and 3, we need values for the parent scale and lower bound $e^{\text{parent}}, u^{\text{parent}}$. We refer to these values as a *traversal context*.

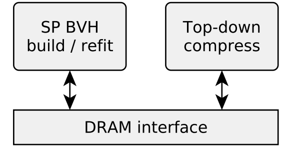
Methods have been proposed to traverse CBVHs at a low arithmetic precision instead of unpacking each node and using single-precision collision tests [Kee14, VAMS16]. In single-precision traversal, the input ray is tested against each AABB with the slabs test [KK86], which first computes parametric distances to each plane,

$$t_{u_i} = (p_i - o_i) d_i^{-1}; \quad t_{v_i} = (q_i - o_i) d_i^{-1}, \qquad (4)$$

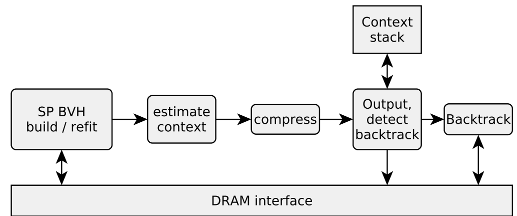
where o, d are the ray origin and direction, and t_{u_i} and t_{v_i} are the parametric distances to the lower and upper bound planes on axis i, respectively. The distances are then combined with *min* and *max* operations to form parametric near and far distances (t_{near}, t_{far}) to the AABB. If $t_{far} < t_{near}$, the ray does not intersect the AABB. The most recent low-precision approach [VAMS16] instead computes plane distances incrementally during traversal:

$$t_{lb_i} = t_{lb_i}^{\text{parent}} + r 2^{e_i^{\text{parent}}} d_i^{-1}; \quad t_{ub_i} = t_{ub_i}^{\text{parent}} - s' 2^{e_i^{\text{parent}}} d_i^{-1}. \qquad (5)$$

As a result, single-precision multiplications can be replaced with cheaper 24×6 bit multiplications. In order to guarantee watertight traversal, the quantized offset s' is computed from the parent upper bound rather than the lower bound, and rounding modes are selected to maximize t_{far} and minimize t_{near}.



(a) Postprocessing, top-down compression HW



(b) Streaming, bottom-up compression HW (proposed)

Figure 2: *Hardware organizations for two CBVH compression strategies.*

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

4. Algorithm

In this section, we describe the basic idea of streaming bottom-up CBVH update, followed by techniques to reduce the backtracking memory traffic inherent in the approach.

4.1. Bottom-up construction and backtracking

Top-down compression of a quantized tree is straightforward by evaluating Equation 1. In bottom-up compression, however, a traversal context is unavailable, and has to be estimated. A stream of BVH node pairs in depth-first, bottom-up order can then be compressed into a CBVH as in Algorithm 1. A traversal context is estimated for each processed BVH node and placed on a *traversal context stack*. When processing an inner node, the contexts of its children can be found at the top of the stack. By decoding the newly generated node and comparing the produced child traversal contexts to the stored contexts, we may detect whether a child was invalidated by the new node, i.e., whether the context estimated when encoding the child was incorrect. The algorithm then recursively backtracks to repair the invalidated children (function `Backtrack` in Algorithm 1). Figure 2b shows a possible hardware organization implementing Algorithm 1, compared to separate top-down compression in Figure 2a.

Context estimation (function `EstimateContext`) is straightforward for e^{parent} by examining the exponent of $q-p$. For u^{parent} it is difficult with the formulation of [VAMS16] where the toplevel grid is aligned to the uncompressed lower bound of the scene, as this is unknown until the end of bottom-up traversal. We instead align all grids to zero, i.e.,

$$u_i = k2^{e_i^{parent}} \quad ; \quad v_i = l2^{e_i^{parent}} k,l \in \mathbb{Z}. \tag{6}$$

By virtue of this alignment scheme, we can estimate u^{parent} by simply rounding p^{parent} down to the nearest multiple of $2^{e_i^{\text{parent}}}$

Each backtracking iteration decodes the target compressed node with its original context and recodes the single-precision bounds with a new context. Child contexts are then checked to determine whether to continue recursion further.

The approach so far is able to produce correct trees. However, backtracking may cause enough memory traffic to undo the savings from streaming compression. It should be noted that backtracking is more expensive than the original encoding, and makes random accesses to the memory, while the main streaming compression algorithm has a more efficient linear access pattern. In our initial experiments on bottom-up update, backtracking almost completely eliminates the memory traffic savings from streaming compression. Hence, it is interesting to minimize backtracking. We next investigate approaches to reduce backtracking.

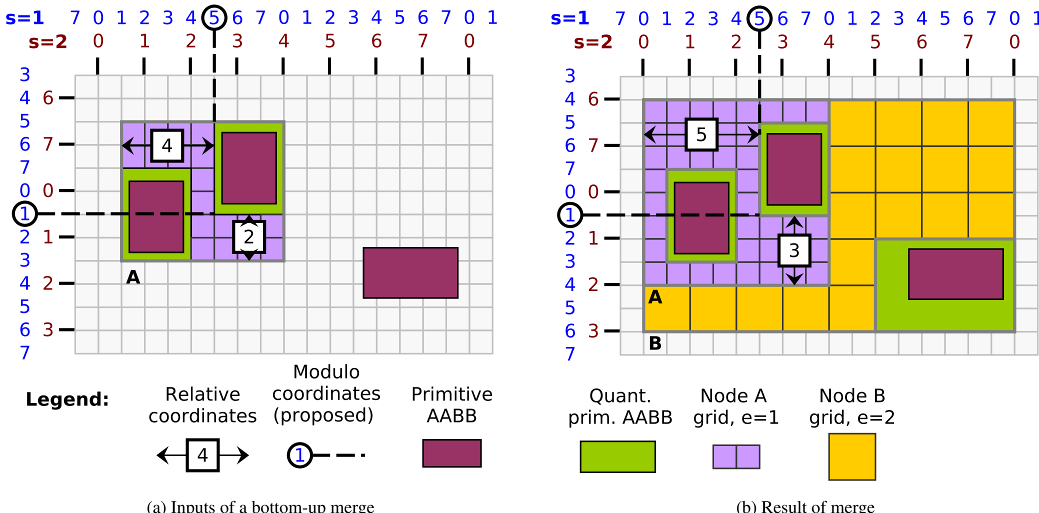
Algorithm 1: Bottom-up streaming CBVH compression algorithm

```
    Data: nodes
    Data: nodeCount
    Data: contextStack
 1  Function Backtrack(idx, oldContext, newContext) is
 2      oldChildContexts[]
 3          ← Decode ( nodes[idx], oldContext ) ;
 4      nodes[idx], newChildContexts[]
 5          ← Encode ( oldChildContexts[0].aabb,
 6                     oldChildContexts[1].aabb,
 7                     newContext) ;
 8      foreach i ∈ 1,2 do
 9          if (oldChildContexts[i].scale ≠
             newChildContexts[i].scale) and (child is not a leaf)
             then
10              Backtrack(nodes[idx].ptr[i],
11              oldChildContexts[i], newChildContexts[i]) ;
12          end
13      end
14  end
15  Function BUCompressNode(bvhNode, contextStack) is
16      context ← EstimateContext(bvhNode) ;
17      nodes[nodeCount], childContexts[]
18          ← Encode ( bvhNode.aabbs[0],
19                     bvhNode.aabbs[1],
20                     context) ;
21      nodeCount ← nodeCount + 1 ;
22      foreach child ∈ 2,1 do
23          if child is not a leaf then
24              storedContext ← contextStack.Pop() ;
25              if childContexts[child].scale ≠ storedContext.scale
                 then
26                  Backtrack(bvhNode.ptr[child],
                     storedContext, childContexts[child]) ;
27              end
28          end
29      end
30      contextStack.Push(context) ;
31  end
32  Function BUCompress(bvhTreeInBUOrder) is
33      contextStack ← empty stack ;
34      foreach node ∈ bvhTreeInBUOrder do
35          BUCompressNode(node, contextStack) ;
36      end
37  end
```

4.2. Modulo encoding

When applying the above compression scheme to test scenes, we note that child nodes are often invalidated even though their decompressed bounds are unchanged. We examine a typical case in Fig. 3, in which a node A is combined with a primitive to form a new node B. When encoding A, a scale $s=1$ is used. When encoding B, the bounds of A are snapped to a grid with scale $s=2$ and rounded. Though the child coordinates encoded by A still refer to the same coordinates, they are relative to the node bounding box derived from B, which has shifted. We find that in real scenes a majority of backtracking is due to this type of context mismatch. It is, then, interesting to find an encoding where the low-precision coordinates can represent the same points as the relative coordinates, but are robust to small changes in parent bounds.

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.



(a) Inputs of a bottom-up merge

(b) Result of merge

Figure 3: *Example of a bottom-up CBVH merge of a primitive with a 2-primitive node. 3 bit coordinates are used, i.e., a node can be up to 8 internal gridcells wide. After snapping the child node bounds to the new parent's grid, its relative coordinates are invalidated. The proposed modulo coordinates are more robust and only invalidated by a parent scale change.*

We describe here a novel encoding which has this desired property, and follows from the global grids described earlier. Given that

$$u_i = k2^{e_i^{parent}}; \qquad v_i = l2^{e_i^{parent}} k,l \in \mathbb{Z}, \tag{7}$$

we can replace (r,s) used in Equations 1 and 3 with modulo coordinates $(\hat{r},\hat{s})$ such that

$$\hat{r} = k \mod 2^N; \qquad \hat{s} = l \mod 2^N, \tag{8}$$

where mod is the integer modulo operation, i.e., $a = b\lfloor a/b \rfloor + (a \mod b)$. To traverse a modulo encoded tree, given the parent's lower-bound modulo coordinate in the local grid $\hat{r}^{parent}$, we can recover relative coordinates as

$$\begin{aligned} r &= (\hat{r} - \hat{r}^{parent}) \mod 2^N; \\ s &= \begin{cases} (\hat{s} - \hat{s}^{parent}) \mod 2^N & \text{if } \hat{r} \neq \hat{s}, \\ (\hat{s} - \hat{s}^{parent}) \mod 2^N + 2^N & \text{if } \hat{r} = \hat{s}. \end{cases} \end{aligned} \tag{9}$$

Note that parent lower bound modulo coordinates r^{parent} are needed for traversal, i.e., they are added to the traversal context. As with relative encoding, as long as the parent bounds limit a range of up to 2^N gridcells in the local scale, modulo coordinates can encode any child range. However, the modulo encoding *is robust to changes in parent bounds as long as the parent scale is unchanged.* Moreover, while relative encoding stores $s_i - 1$ to avoid overflow, $\hat{s}$ can be stored as is.

C pseudocode for traversal is shown in Fig. 4. Note that the parent modulo coordinate needs to be scaled to the local grid before use (line 1). In the code, floating-point coordinates are recovered (lines 8–9), but we could instead use the relative coordinates directly for ray tracing as in Vaidyanathan's [VAMS16] work. In this case, the upper bound coordinate relative to parent upper bound, s', is recovered as

$$s' = (\hat{s}^{parent} - \hat{s}) \mod 2^N, \tag{10}$$

and the parent upper bound s_{parent} needs to be added to the traversal context. The grid scale is saturated close to the minimum representable floating point number (lines 17).

We next discuss a hardware-friendly way to compress a given floating point node pair to modulo encoding. C pseudocode for compression, along one axis, is shown in Fig. 5. First, the input floating point upper and lower bounds are broken into sign, mantissa and exponent (lines 1–5). They are then aligned to local grid scale, rounding both coordinates down. (lines 7–8). Next the mantissas are converted from sign-magnitude representation to two's complement (lines 10–12). The upper bound is incremented, ensuring that the quantized upper bound is not lower than the original (line 14). The low mantissa bits now correspond to the quantized modulo coordinates $\hat{r},\hat{s}$ and are extracted (lines 16–18). Note that the input scale must be selected such that at this point `ub_m` $-$ `lb_m` $\leq 2^N$. We now have enough information to store the compressed node in memory. In a bottom-up build, we next need to decide whether to backtrack, and compute traversal contexts for backtracking to each child. Backtracking detection is based on the child scale, which is computed next (lines 20–28). In addition to the scale and child scale, the context includes `lb_mod`, and the decompressed child AABB (`clb_out, cub_out`) (computed in lines 30–39). Note that `lb` cannot be substituted for `clb_out`,

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

```
parent_lb_mod <<= (parent_scale - scale);

int rel_lb = (lb_mod - parent_lb_mod) &
    ((1<<QUANT_BITS)-1);
int rel_ub = (ub_mod - parent_lb_mod) &
    ((1<<QUANT_BITS)-1);
if(lb_mod == ub_mod)
    rel_ub += (1<<QUANT_BITS);

clb_out = parent_clb + glm::ldexp(float(
    rel_lb), scale);
cub_out = parent_clb + glm::ldexp(float(
    rel_ub), scale);

int gridsize = rel_ub - rel_lb;
child_scale_out = scale;
while(gridsize < (1<<(QUANT_BITS-1))) {
    gridsize <<= 1;
    child_scale_out--;
}
if(child_scale_out < -126)
    child_scale_out = -126;
```

Figure 4: *Modulo-encoded CBVH traversal along one axis.* `clb_out`, `cub_out`*: Decompressed lower and upper bound* (u_i,v_i). `lb_mod`, `ub_mod`*: Modulo coordinates of lower and upper bound* $(\hat{r}_i,\hat{s}_i)$. `parent_lb_mod`*: Node lower bound modulo coordinate, in parent scale* $(\hat{r}_i^{parent})$. `parent_scale`, `scale`, `child_scale_out`*: scale* (e_i) *of parent, current and child node.*

since the AABB decoded as in Figure 4 (lines 8–9) is not then guaranteed to contain the uncompressed AABB.

4.2.1. Hardware complexity

The overhead of modulo encoding in traversal consists of low-precision arithmetic. Lines 3, 4, 10 in Figure 4 represent N-bit adders, while lines 13–17 can be implemented with a priority encoder and a low-precision subtractor. The child axis computation logic of lines 10–17 is also present in relative coordinate traversal. In total, the overhead is equal to ca. 2 N-bit adders and 1 shifter per AABB axis, or 12 adders and 6 shifters for a node pair. Using the component costs in [VAMS16], the overhead of the adders is 5% in addition to a *traversal point update* traversal unit, or 13% against a shared-plane traversal unit.

A hardware implementation of a single-axis compressor shares much of its structure with a floating-point adder. Figure 6 contrasts a compressor to the significand datapath of an adder. The main components of an adder are input alignment, significand addition/subtraction, normalization and rounding. In Fig. 5, we align two inputs like the adder equivalent, apply rounding, and normalize back to IEEE-754 single-precision. The cost of conversions to and from two's complement representation is similar to that of rounding. In summary, a single-axis compressor has roughly 2x the input alignment, 2x the normalization, and 6x the rounding logic of a single-precision adder. We can approximate it as the equivalent of

```
//inputs -> sign, exponent, mantissa
int lb_s, lb_e, lb_m;
int ub_s, ub_e, ub_m;
break_float(lb, lb_s, lb_e, lb_m);
break_float(ub, ub_s, ub_e, ub_m);

alignf(lb_e, lb_m, lb_s, scale, RND_DOWN);
alignf(ub_e, ub_m, ub_s, scale, RND_DOWN);

//sign-magnitude -> 2's complement
if(lb_s) lb_m = -lb_m;
if(ub_s) ub_m = -ub_m;

ub_m++;

// Output modulo coordinates
lb_mod_out = lb_m & ((1 << QUANT_BITS)-1);
ub_mod_out = ub_m & ((1 << QUANT_BITS)-1);

// Output child scale
int gridsize = ub_m - lb_m;
child_scale_out = scale;
while(gridsize < (1<<(QUANT_BITS-1))) {
    gridsize <<= 1;
    child_scale_out--;
}
if(child_scale_out < -126)
    child_scale_out = -126;

// Decode bounds
lb_s = (lb_m < 0) ? 1 : 0;
if(lb_s < 0)
    lb_m = -lb_m;
ub_s = (ub_m < 0) ? 1 : 0;
if(ub_s)
    ub_m = -ub_m;

clb_out = pack_float(lb_m, scale, lb_s);
cub_out = pack_float(lb_m, scale, ub_s);
```

Figure 5: *Hardware-oriented algorithm for modulo-encoded CBVH compression.* `lb`,`ub`*: Input uncompressed lower and upper bound* (p_i,q_i). `lb_mod_out`, `ub_mod_out`*: Output modulo coordinates* $(\hat{r}_i,\hat{s}_i)$.

two adders, as significand addition is more expensive than rounding. A BVH node pair compressor has six single-axis compressors and, in a bottom-up build, scale estimators for each axis, which are the equivalent of single-precision subtractors, for a total cost of 15 single-precision adders. This appears a reasonable cost considering that, e.g., the tree builder in [DFM13] has over 500 FPUs.

4.3. Treelet-based compression

We find from experiments with real scenes that backtracking typically only descends one or two hierarchy levels before terminat-

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

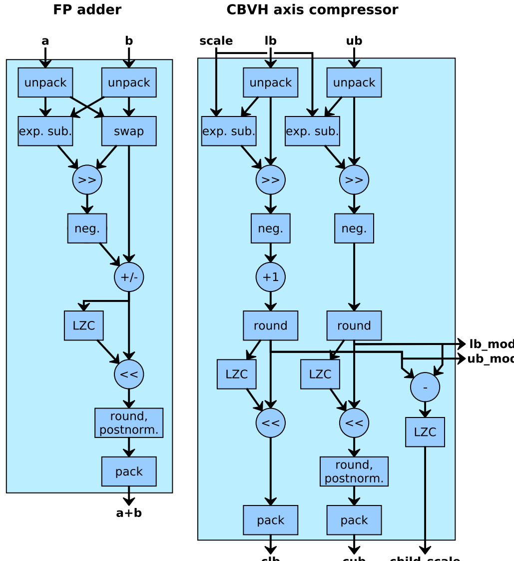


Figure 6: *Hardware implementation of a CBVH compressor (as in Figure 5 based on modulo coordinates, contrasted to a floating-point adder. LZC: leading zero counter. For clarity, only the significand datapath of the FP adder, which comprises most of the logic, is shown. The single-axis compressor has similar computational resources as two FP adders.*

ing. Fig. 7 shows the depth histogram of backtracking iterations in a test scene: it is visible that the distribution is top-heavy. Since the working set for possible, e.g., 1- or 2-level backtracks is easily stored on chip, there are many possible ways to avoid the memory traffic from these short backtracks. We describe one approach here which eliminates backtracking down to a fixed depth while always retaining fully pipelined throughput.

In the proposed approach, we place logic after the compression and update unit to collect BVH nodes into small treelets with a fixed maximum depth M. Each treelet is then compressed top-down. Context estimation is done only at the treelet root node, while only the nodes at level M are output. This is almost equivalent to pre-emptively backtracking all treelets, except there is no need to decompress nodes, simplifying the hardware. A hardware architecture with a depth of 3 is shown in Figure 8. An M-level treelet has up to M^2-1 nodes, so the cost of this approach grows quickly, but it is evident from Figure 7 that removing backtracks of even 1–2 levels gives most of the benefits available. Treelet collection is performed by similar stack-based hardware as backtrack detection: each stack element has space for a treelet of depth $M-1$. Plane sharing [FD09] may be used to compress the treelets and reduce chip area, as the compression and decompression is very cheap in fixed-function hardware.

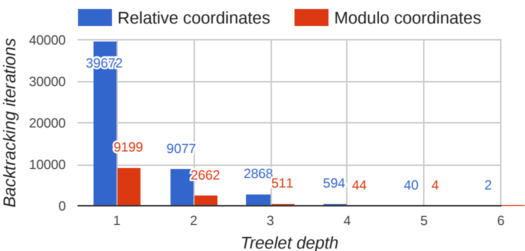


Figure 7: *Depth histogram of backtracking iterations in the Elephant scene with relative and modulo coordinates.*

4.4. Minimum scale adjustment

The techniques proposed so far nearly eliminate backtracking in many practical scenes. However, they have difficulties in the special case where the input BVH has large subtrees where all nodes share a degenerate axis, i.e., they have zero width along an axis. In practice, this arises when the scene has axis-aligned, planar, finely tesselated geometry. In this case, the entire subtree is first encoded bottom-up at the minimum scale, e.g. 2^{-126} - close to the minimum representable floating point number - in Figure 5. On encountering non-degenerate geometry with nonzero width along that scale, the subtree is backtracked, such that the scale gradually approaches the minimum throughout the subtree. Since each backtracking iteration can only decrease scale by a factor of 2^N, a large portion of the subtree needs to be backtracked. Modulo encoding is unhelpful in this case since the scale difference is large, and the distribution of backtracking depth is unfavorable for treeletwise compression.

An example from the **cloth** scene is shown in Fig. 9. This type of geometry is rare in typical scenes, but may easily occur in, e.g., synthetic animated scenes, such as the example. In this work, we approach this type of scene by selecting a minimum scale that is coarse enough to avoid the above issue, but fine enough that tree quality is unaffected. As a drawback, a parameter is added to the construction and traversal algorithms which may need adjustment based on scene size. However, we find later in evaluation that there is a wide margin for the value in practical scenes.

Another approach we experimented with was to snap upper bound coordinates up to numbers which are higher than the lower bound and exactly representable in single-precision floating point. For example, the degenerate geometry in the **cloth** scene lies on the plane $z=-0.4$. The corresponding primitive bounding boxes would, then, be $\mathrm{ulp}(-0.4)=2^{-25}$ wide, where ulp() is the *unit in the last place* function. This has similar effects as a manageable minimum scale. This approach avoids adding a parameter, however, it does not work with degenerate geometry where the corresponding coordinate is zero.

4.5. Plane sharing

The methods described so far assume that 6 coordinates are stored per bounding box. It can be beneficial in terms of memory footprint to further compress the tree by reusing coordinates from the parent

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

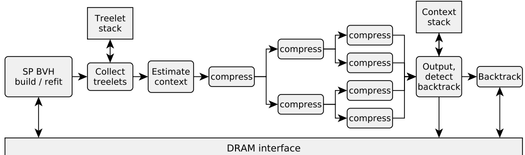


Figure 8: *Hardware architecture for treelet compression*

node, i.e., plane sharing [FD09]. This technique can also be used in CBVHs to reduce the arithmetic cost of traversal [VAMS16]. In this scheme, 6 coordinates are stored per a pair of bounding boxes, and the remaining 6 are reused from the parent AABB. Extra bits are included in the node datastructure to denote which coordinates are reused. However, unlike full-precision BVH, in CBVH this is a lossy process: a coordinate quantized to a coarse scale at a high hierarchy level often differs from the same coordinate quantized to a finer scale. This causes only modest quality loss, but greatly complicates bottom-up construction, as it often invalidates nodes far down in the hierarchy. So far, we do not have a satisfactory solution for plane sharing, and it is left as an open problem.

5. Evaluation

In order to evaluate the proposed methods, we implemented the proposed streaming bottom-up compression algorithms, as well as baseline top-down compression, in software. The algorithms are evaluated for LBVH construction and refitting, assuming hardware units like [VKJ*15, NKP*15]. A set of 16 test scenes is used for evaluation, as listed in Figure 11. Common assumptions about input and output data layouts are as follows. Input primitive data is stored as an array-of-structures of triangles with three vertices (á 36B) and a primitive ID (4B) per triangle. We assume a CBVH format with a memory footprint of 16B per node pair, which is the same density as in [Kee14]. Further space could be saved by optimizing the child pointer representation as in, e.g., [SE10, LV16], but this is outside the scope of this work. Each node in a pair has six 6-bit relative or modulo coordinates and a leaf bit, which leaves space for a 27-bit child pointer. Leaf pointers in the CBVH index a primitive ID array as in, e.g., [Wal07], which references the input primitive data. LBVH needs to output the ID array, while refitting leaves it unchanged.

5.1. Minimum scale

We first calibrate the minimum scale parameter to be used in later benchmarks. As discussed in Subsection 4.4, the minimum scale should be coarse enough to avoid issues with axis-aligned, planar geometry. However, an overly coarse scale has an adverse effect on tree quality. The measured tradeoff is illustrated in Figure 10. In our set of benchmarks, only **cloth** has a significant amount of thin geometry, so we examine **cloth** and the other scenes separately. With a fine minimum scale, **cloth** exhibits a large amount of backtracking, but this is mostly eliminated at a minimum scale of 2^{-30}. Conversely, up to a minimum scale of 2^{-15}, there was no effect on scene quality. Therefore, there appears to be a wide margin for the

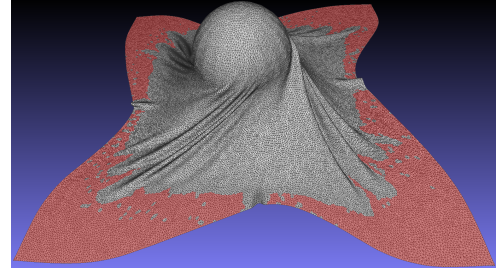


Figure 9: *Example scene (**cloth**) with a large amount of geometry with a degenerate axis (red).*

minimum scale parameter. In the following benchmarks, we use a value of 2^{-30}. It should be noted that though axis-aligned geometry is often found in indoor and architectural scenes, several such scenes were included in the benchmark (**crytec**, **conference**, **italian**, **babylonian**), but did not cause significant backtracking even without minimum scale adjustment. Axis-aligned geometry, therefore, only seems to become an issue in synthetic worst-case scenes.

5.2. Backtracking and memory traffic

Memory traffic is computed based on the numbers of primitives, nodes and backtracking iterations recorded from the software builder. For LBVH, we assume the hardware builder [VKJ*15] is modified to output large leafs when multiple primitives share the same Morton code. The builder reads input primitives (40B traffic per primitive), sorts their AABBs (2x32B per primitive), and outputs a CBVH hierarchy (16B per node) and primitive ID array (4B per primitive). For refitting, we update LBVH trees to match the original geometry (as the exact geometry has no bearing on memory traffic). Refitting is assumed to proceed by Wald's algorithm [WBS07], i.e., it needs to read the CBVH hierarchy (16B per node), primitive ID array (4B per primitive) and primitives (40B per primitive), and output an updated CBVH hierarchy (16B per node). In the case of one primitive per leaf, this gives the memory traffic in Fig. 1. Finally, each backtracking iteration with the bottom-up builds requires a minimum-size DRAM read and write. We assume a DRAM interface with a 64B access granularity, e.g., a LPDDR4 interface with a 64-bit channel.

Table 1 shows results in aggregate and Table 3 per scene. We see that modulo encoding gives roughly the same benefits as one level of treeletwise compression: they reduce backtracking by 4.4x and 6.8x, respectively. The gains from modulo encoding are orthogonal with treeletwise compression: together, they reduce backtracking by 15x. With two-level treelet compression or one-level compression combined with modulo encoding, the amount of backtracking is so low that the memory traffic results are close to ideal.

The main scene parameter affecting memory traffic savings is the number of primitives per leaf: the worst-case **conference** scene has 8.7 triangles per leaf, while most scenes have 1–2. It should be noted that some LBVH builders store one primitive per leaf,

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

Table 1: *Normalized backtracking and memory traffic results with (baseline) postprocessing top-down compression and streaming bottom-up compression, with combinations of proposed modulo coordinates and treelet-based compression (with different treelet depths). Backtracking results are normalized to baseline bottom-up compression (relative encoding without treelets). Memory traffic is normalized to top-down compression.*

Compression dir.		Top-down	Bottom-up (proposed)							
Coordinates		Relative					Modulo (proposed)			
Treelet depth		-	-	2	3	4	-	2	3	4
Backtrack iterations	Min	-	1.00	0.06	0.01	0.00	0.10	0.02	0.00	0.00
	Max	-	1.00	0.25	0.09	0.03	0.44	0.15	0.06	0.03
	Avg	-	1.00	0.15	0.04	0.01	0.22	0.06	0.02	0.01
Memory traffic (LBVH)	Min	1.00	0.75	0.53	0.51	0.50	0.54	0.51	0.50	0.50
	Max	1.00	0.95	0.86	0.84	0.83	0.87	0.84	0.83	0.83
	Avg	1.00	0.84	0.61	0.59	0.58	0.63	0.59	0.58	0.58
Memory traffic (Refit)	Min	1.00	0.67	0.41	0.39	0.38	0.42	0.39	0.38	0.38
	Max	1.00	0.90	0.74	0.70	0.69	0.76	0.71	0.69	0.68
	Avg	1.00	0.79	0.49	0.45	0.44	0.52	0.46	0.44	0.44

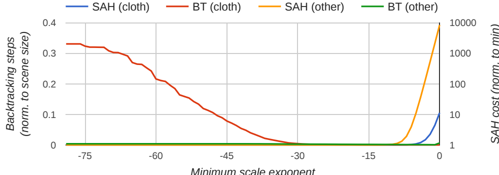


Figure 10: *Effect of minimum scale on tree quality and backtracking: **cloth** scene and average of other scenes. Tree quality is represented by SAH cost normalized to minimum value. The number of backtracking steps is normalized to scene size. The planar geometry in **cloth** begins to cause significant backtracking with a minimum scale of less than 2^{-30}. An overly coarse minimum scale, ca. 2^{-10}, harms tree quality.*

e.g. [Kar12]. For this type of tree, streaming construction would consistently give best-case savings.

In LBVH, plain streaming bottom-up construction saves 16% memory traffic compared to the baseline of BVH output and postprocessing compression. Modulo coding improves savings to 37%, a single level of treelet compression to 39%, and a combination of both techniques to 41%. At this point backtracking traffic is small, so adding two more levels of treelet compression improves savings only to 42%, even as backtracking falls 10x. In refitting the savings are larger: on average 21% without either backtracking optimization, 54% with both optimizations, and 56% with 4-level treelet compression.

Since modulo encoding adds some overhead to traversal, and similar memory traffic savings can be recovered by adding a level of treelet compression which only complicates the update, it may be advantageous to store the CBVH in relative coordinates and rely on treelet compression. In this case, modulo encoding is used to enable inexpensive compression hardware as in Figure 6, though the resulting nodes are immediately converted to relative representation.

Table 2: *LBVH runtime results (ms) for single-precision build, postprocessing compression, and streaming compression with and without proposed optimizations.*

Output	BVH	CBVH			
Compression dir.	-	TD	BU (proposed)		
Coordinates	Float	Rel.	Rel.	Mod.	Leaf size
Treelet depth	-	-	-	4	
Toasters	0.2	0.2	0.3	**0.1**	1.1
Bunny	1.1	1.6	1.9	**0.8**	1.1
Elephant	1.3	1.9	2.2	**1.0**	1.1
Cloth	1.5	2.1	2.4	**1.1**	1.0
Fairy	2.1	2.5	2.6	**1.9**	3.8
Armadillo	3.4	4.8	5.7	**2.6**	1.2
Crytek	3.6	4.8	5.4	**3.0**	2.0
Conference	3.2	3.5	3.4	**2.9**	8.7
Sportscar	4.3	5.8	6.4	**3.5**	1.9
Italian	5.0	6.6	7.7	**4.2**	2.8
Babylonian	6.7	8.8	10.2	**5.6**	2.4
Hand	9.6	13.3	15.2	**7.6**	1.5
Dragon	13.4	19.0	21.5	**10.3**	1.3
Buddha	16.7	23.5	26.0	**13.0**	1.5
Lion	23.9	33.6	37.7	**18.7**	1.3
Hairball	40.9	56.5	52.2	**32.2**	1.4
Average	-	+36%	+52%	-20%	2.1

5.3. Runtime

We further examine the runtime effects of the proposed techniques on LBVH construction by means of architectural simulations, based on cycle-level simulation of the LBVH builder. Four alternatives are compared: single-precision BVH build, postprocessing top-down compression, streaming compression, and streaming compression with the proposed optimizations. In the last alternative, modulo encoding is used and treelet depth is set at 4. For a conservative comparison, we use optimistic assumptions for the postprocessing compression hardware used as baseline, and pessimistic assumptions for the backtracking unit required by the proposed method. The backtracking unit performs backtracking steps sequentially, while the compression unit starts each memory oper-

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

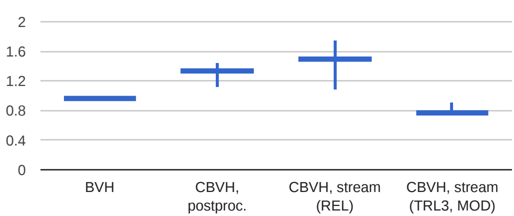


Figure 11: *LBVH runtime results, normalized to single-precision LBVH. Average values and ranges are shown.*

ation immediately after its dependencies are available, corresponding to a highly parallel, multi-threaded implementation. The LBVH builder is configured with a 256KB scratchpad memory and a buffer size of 8 AABBs, such that it can handle scenes of up to 4M triangles. The operating frequency is set at 1GHz. We assume a dual-channel, 32-bit LPDDR3-1600 memory interface with a maximum bandwidth of 12.8GB/s, which is simulated with the cycle-accurate Ramulator model [KYM15].

Runtime results are shown in Table 2 and Figure 11. As expected from the memory traffic results, the proposed method is consistently faster than single-precision BVH construction, while postprocessing compression is slower. On average, the proposed methods are 20% faster than single-precision LBVH, while baseline post-processing top-down compression is 36% slower.

5.4. Tree quality

In order to verify the quality of the constructed trees, we ray traced each test scene and counted intersection tests. CBVHs required, on average, 8% more box tests and 13% more triangle tests than BVH, in line with previous work [Kee14, VAMS16]. Out of the proposed techniques, minimum scale adjustment affects quality as described in Section 5.1. Moreover, context estimation sometimes gives more loose scales than top-down build, resulting in a nonzero but negligible quality difference (<0.1% for all scenes). Modulo encoding and treelet-based compression had no effect on quality.

5.5. Watertightness

In addition to performance and tree quality, it is interesting whether a ray tracing system is watertight, i.e., whether it is guaranteed to avoid false misses. Vaidyanathan's traversal algorithm [VAMS16] is proven watertight based on the criterion that exact parametric distances t_{max}, t_{min} to each AABB are enclosed in the distances computed during traversal. We do not have a formal proof that our trees satisfy the criterion, but verified that this criterion held over every box test when path tracing all test scenes. Moreover, decompressing our bounds to single precision always gave AABBs that enclose the uncompressed AABB.

6. Limitations and Future Work

A main limitation of the present work is that shared-plane CBVHs [VAMS16] cannot yet be updated, and are left as an open problem. It may also be interesting to use shallow hierarchies as in [VKJT16] instead of shared-plane encoding, as this gives at least some of the memory footprint advantage of shared-plane encoding while making bottom-up updates more tractable. Moreover, since the node size of shared-plane CBVHs is small compared to cache lines, only a small fraction of the memory footprint advantage translates into memory bandwidth savings [VAMS16], at least in a straightforward implementation. From this perspective, it may be advantageous to use a shallow hierarchy and traverse fewer, larger nodes.

In this work, full-precision child pointers were used for simplicity, but it seems possible to integrate, e.g., the techniques of Liktor et al. [LV16] or Segovia et al. [SE10] to compress pointer fields. Moreover, using compact primitive storage formats such as triangle strips [LYM07] may be a low-hanging fruit to speed up tree update. Although this article focused on fixed-function hardware, the proposed encoding may also be interesting for CBVH builds on a programmable GPU, where bottom-up algorithms are preferred in order to take advantage of the GPU's parallel resources.

7. Conclusion

This article showed that the construction and refitting of compressed BVH trees can be significantly optimized by means of streaming, bottom-up compression. Two novel techniques were proposed to enable the rapid hardware-accelerated update of compressed BVHs in animated scenes: modulo encoding and treelet-based compression. Together, the proposed techniques allow streaming compression in bottom-up order, reducing memory traffic from LBVH construction by 38% and from refitting by 54% on average, compared to a postprocessing compression step. Since both LBVH and refitting are memory-bound algorithms, these savings translate directly into improvements in performance and energy efficiency, and are especially significant on mobile devices with limited power budget and memory bandwidth. Runtime was evaluated for LBVH builds, and was in line with the memory traffic reductions. Notably, it is faster to update CBVHs than uncompressed BVHs with the proposed method. The described work represents a step toward real-time, mobile ray tracing of large-scale animated scenes.

Acknowledgement

The authors would like to thank the anonymous reviewers for their insightful comments. This work was financially supported by the TUT graduate school and the Academy of Finland (decision #297548, PLC). The 3D models used are courtesy of Ingo Wald (Fairy), Andrew Kensler (Toasters), Yasutoshi Mori (Sportscar), Frank Meinl (Crytek Sponza), Jonathan Good (Italian, Babylonian), Anat Grynberg and Greg Ward (Conference), Naga Govindaraju, Ilknur Kabul and Stephane Redon (Cloth), the Stanford Computer Graphics Laboratory (Bunny, Buddha, Dragon), and Samuli Laine (Hairball). Crytek Sponza and Dragon have modifications courtesy of Morgan McGuire [McG11].

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

References

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)* (2014). 2

[DFM13] DOYLE M., FOWLER C., MANZKE M.: A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics 32*, 4 (2013), 139:1–10. 1, 3, 6

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics* (2015), pp. 13–20. 2

[FD09] FABIANOWSKI B., DINGLIANA J.: Compact bvh storage for ray tracing and photon mapping. In *Proc. Eurographics Ireland Workshop* (2009), pp. 1–8. 2, 7, 8

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics* (2011), pp. 59–64. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *Computer Graphics and Applications 7*, 5 (1987), 14–20. 2

[IWP07] IZE T., WALD I., PARKER S. G.: Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proc. Eurographics Conf. Parallel Graphics and Visualization* (2007), pp. 101–108. 3

[Kar12] KARRAS T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. High-Performance Graphics* (2012), pp. 33–37. 2, 9

[Kee14] KEELY S.: Reduced precision hardware for ray tracing. In *Proc. High-Performance Graphics* (2014), pp. 29–40. 1, 2, 3, 8, 10

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, effective BVH updates for animated scenes. In *Proc. Symp. Interactive 3D Graphics and Games* (2012), pp. 197–204. 1, 3

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *ACM SIGGRAPH Computer Graphics* (1986), vol. 20, ACM, pp. 269–278. 3

[KYM15] KIM Y., YANG W., MUTLU O.: Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters PP*, 99 (2015), 1–1. 10

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 1, 2

[LSL*13] LEE W., SHIN Y., LEE J., KIM J., NAH J., JUNG S., LEE S., PARK H., HAN T.: SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. High-Performance Graphics* (2013), pp. 109–119. 1

[LV16] LIKTOR G., VAIDYANATHAN K.: Bandwidth-efficient BVH layout for incremental hardware traversal. In *Proc. High Performance Graphics* (2016), pp. 51–61. 1, 8, 10

[LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2007), IEEE, pp. 19–26. 10

[McG11] MCGUIRE M.: Computer graphics archive, 2011. http://graphics.cs.williams.edu/data/meshes.xml. 10

[MW06] MAHOVSKY J., WYVILL B.: Memory-conserving bounding volume hierarchies with coherent raytracing. In *Computer Graphics Forum* (2006), vol. 25, pp. 173–182. 2

[NKK*14] NAH J., KWON H., KIM D., JEONG C., PARK J., HAN T., MANOCHA D., PARK W.: RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics 33*, 5 (2014), 162:1–15. 1

[NKP*15] NAH J., KIM J., PARK J., LEE W., PARK J., JUNG S., PARK W., MANOCHA D., HAN T.: HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (2015), 389–401. 1, 3, 8

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High-Performance Graphics* (2010), pp. 87–95. 2

[Pow15] POWERVR: PowerVR ray tracing. Accessed Feb 20, 2017, 2015. http://imgtec.com/powervr/ray-tracing/. 1

[SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proc. Graphics Interface* (2010), pp. 153–160. 2, 8, 10

[SKBD12] SPJUT J., KOPTA D., BRUNVAND E., DAVIS A.: A mobile accelerator architecture for ray tracing. In *Proc. Workshop on SoCs, Heterogeneous Architectures and Workloads* (2012). 1

[SKKB09] SPJUT J., KENSLER A., KOPTA D., BRUNVAND E.: TRaX: a multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28*, 12 (2009), 1802–1815. 1

[VAMS16] VAIDYANATHAN K., AKENINE-MÖLLER T., SALVI M.: Watertight ray traversal with reduced precision. *Proc. High-Performance Graph* (2016). 2, 3, 4, 5, 6, 8, 10

[VKJ*15] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., KULTALA H., TAKALA J.: MergeTree: a HLBVH constructor for mobile systems. In *SIGGRAPH Asia Technical Briefs* (2015), p. 12. 1, 2, 3, 8

[VKJT16] VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., TAKALA J.: Multi bounding volume hierarchies for ray tracing pipelines. In *SIGGRAPH Asia Technical Briefs* (2016), p. 8. 10

[Wal07] WALD I.: On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2007), pp. 33–40. 2, 8

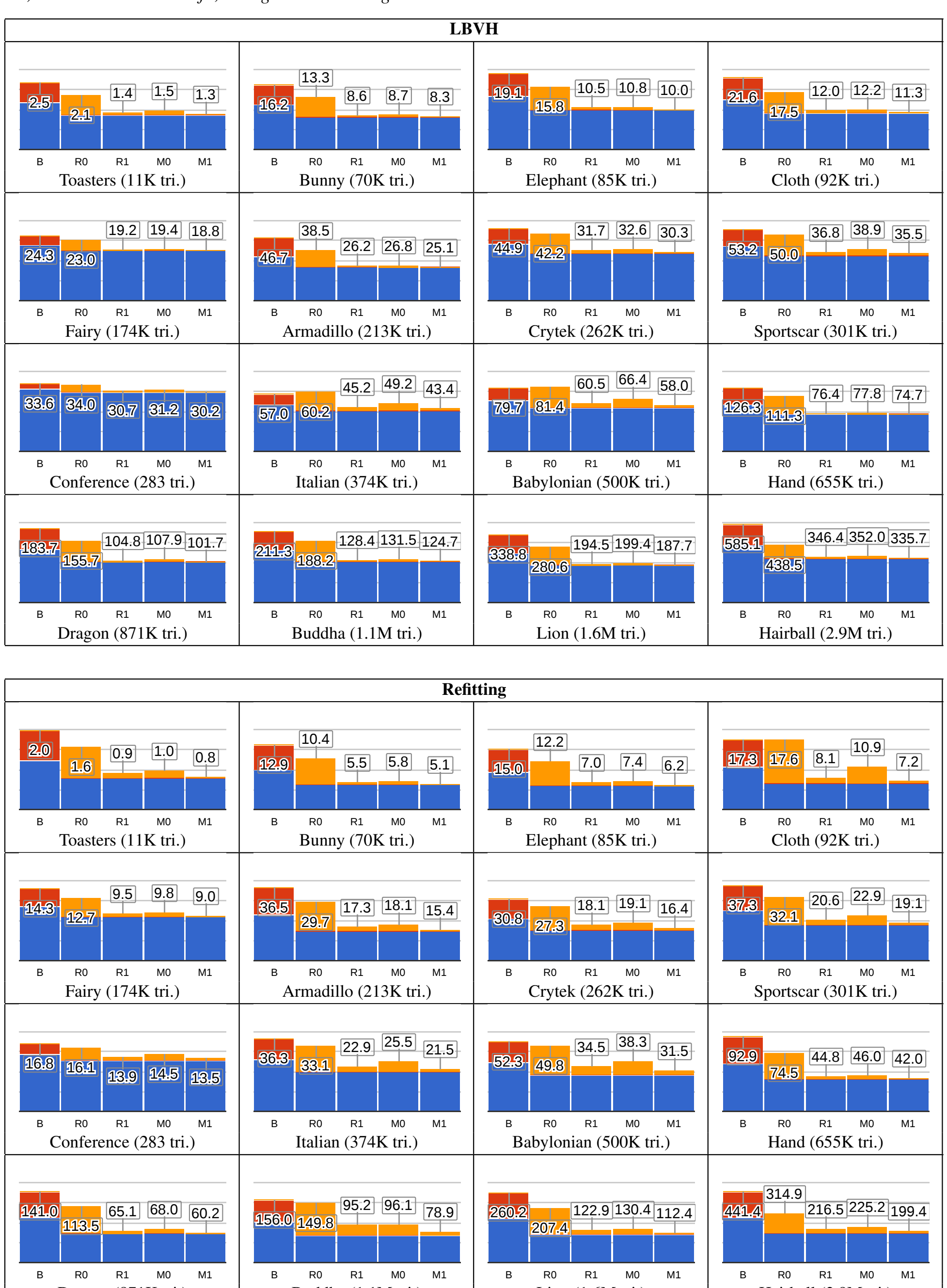
[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2008), IEEE, pp. 49–57. 1

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6. 2, 3, 8

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444. 1

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

Table 3: *Memory traffic with LBVH and refitting. B: Baseline, separate top-down compression. R0: Relative encoding. R1: Relative encoding, 1 layer of treelet-based compression. M0: Modulo encoding. M1: Modulo encoding, 1 layer of treelet-based compression. Red: Separate compression, blue: construction/refit, orange: backtracking.*



© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd.

[P4] Publication 4

[P4] Viitanen T., Koskela M., Jääskeläinen P., Takala J.:
Multi Bounding Volume Hierarchies for Ray Tracing Pipelines.
In *SIGGRAPH Asia Technical Briefs* (2016), p. 8.

Multi Bounding Volume Hierarchies for Ray Tracing Pipelines

Timo Viitanen * Matias Koskela Pekka Jääskeläinen Jarmo Takala

Tampere University of Technology, Finland

Abstract

High-performance ray tracing on CPU is now largely based on Multi Bounding Volume Hierarchy (MBVH) trees. We apply MBVH to a fixed-function ray tracing accelerator architecture. According to cycle-level simulations and power analysis, MBVH reduces energy per frame by an average of 24% and improves performance per area by 19% in scenes with incoherent rays, due to its compact memory layout which reduces DRAM traffic. With primary rays, energy efficiency improves by 15% and performance per area by 20%.

Keywords: ray tracing, ray tracing hardware

Concepts: •**Computing methodologies** → **Ray tracing;** *Graphics processors;* Computer graphics;

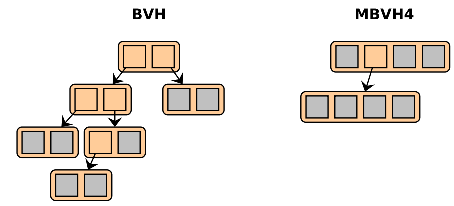


Figure 1: *Left side: BVH organizing 7 leafs (grey) with six 64B nodes (each storing AABBs and pointers of its two children). Right side: 4-wide MBVH organizing the same number of leafs with two 128B nodes ($\frac{2}{3}$ memory footprint).*

1 Introduction

Ray tracing is a fundamental rendering technique which is widely used in offline rendering to model the physical transport of light. Rendering interactive scenes with ray tracing is a longstanding research challenge in computer graphics. In recent years, there has been an influx of research on specialized ray tracing hardware architectures to enable such interactive rendering. Many hardware architectures have been proposed in academic forums, such as RPU [Woop et al. 2005], SGRT [Lee et al. 2013] and RayCore [Nah et al. 2014]. In addition, recently a commercial mobile GPU IP with ray tracing support has been launched. Many of these works are aimed at mobile devices partly since the ray tracing algorithm is well suited for smaller displays, and also because it is likely commercially easier to incorporate a ray tracing feature into a mobile SoC than to sell a stand-alone product. The focus on mobile systems, and recent trends in CMOS process technology place restrictions on ray tracing hardware architectures. Handsets and tablets operate under strict power constraints since they are battery-powered and passively cooled; it is therefore crucial to optimize the hardware accelerator for low-energy operation. Due to logic scaling, the energy cost of computational logic is falling relative to long-range communication. Especially accesses to off-chip SDRAM main memory are expensive.

The basic operation in ray tracing is *ray traversal*, i.e., finding the closest point of the scene geometry which intersects a given half line. The basis of a modern high-performance implementation is to organize the scene geometry into an acceleration datastructure such as a *Bounding Volume Hierarchy* (BVH), which reduces this into a logarithmic-time operation in the typical case. Most of the computational effort in ray tracing goes to traversing this acceleration tree and performing intersection tests against the scene geometry in the leaf nodes. Consequently, ray tracing hardware architectures tend to include fixed-function hardware pipelines for these two tasks.

In this work, we investigate using *Multi-Bounding Volume Hierarchies* (MBVH) [Ernst and Greiner 2008] [Wald et al. 2008] [Dammertz et al. 2008] in a ray tracing accelerator: this is a variant of BVH with a higher branching factor, typically 4. MBVH was originally intended to take advantage of SIMD instruction sets such as SSE in CPUs, but the technique also has general benefits:

- MBVH has a more compact memory layout than BVH, as noted by Dammertz et al. [Dammertz et al. 2008] and illustrated in Figure 1. Consequently, it improves the hit rate of caches and reduces external memory traffic.
- MBVH ray traversal with a 4-wide MBVH (MBVH4) performs roughly the same amount of computation and memory requests as a BVH, but organized into larger consecutive units. $\frac{n}{2}$ random accesses of $2m$ bytes can be served by a simpler memory hierarchy than n accesses of.m bytes. Likewise, organizing computation into larger units, with fewer branches, reduces the overhead of control logic in the architecture.

Recently, Guthe [2014] found MBVHs advantageous in GPU ray tracing, demonstrating the above advantages. In this paper, we show with simulations and power analysis that introducing MBVH4 to a ray tracing unit significantly improves area- and energy-efficiency over a BVH baseline, especially with incoherent rays.

2 Related Work

Few works on ray tracing hardware architecture have variations on the applied data structures; most of the recent approaches focus on plain BVHs and k-d trees. There is recent interest in quantized BVHs with, e.g., 5 bits per coordinate, most recently by Vaidyanathan [2016]. This structure achieves a very high simulated performance. However, it appears nontrivial to keep updated when rendering dynamic scenes, whereas the present work can use similar update and construction methods as a conventional BVH, with minor modifications.

*e-mail:timo.2.viitanen@tut.fi

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

SA '16 Technical Briefs, December 05 - 08, 2016, , Macao
ISBN: 978-1-4503-4541-5/16/12
DOI: http://dx.doi.org/10.1145/3005358.3005384

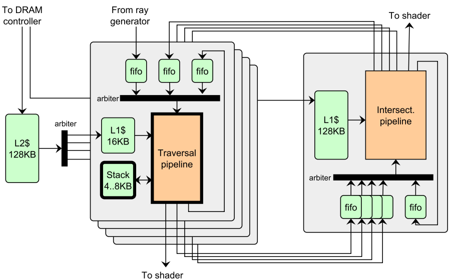


Figure 2: *Ray tracing accelerator architecture modeled in this paper, based on [Lee et al. 2014]. This paper makes changes to the bolded components: the traversal pipeline and the traversal stack.*

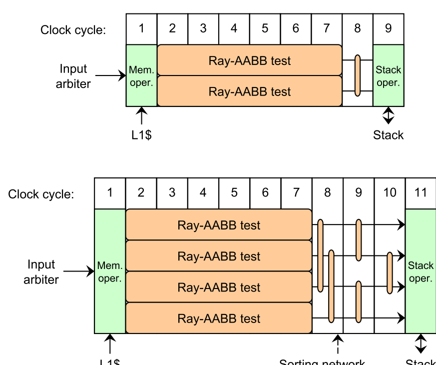


Figure 3: *Top: Baseline TRV unit with two parallel ray-AABB tests. Bottom: Proposed MBVH TRV unit with four ray-AABB tests and sorting network. Pipeline stages are numbered on top.*

Lee et al. [Lee et al. 2014] optimize the BVH data layout in the SGRT system [Lee et al. 2013] for a large performance gain. Hwang et al. [Hwang et al. 2015], optimize the number representations in their data structures to mostly fixed-point-arithmetic, but resort to floating-point numbers in cases where they are more efficient in terms of hardware cost. Our work introduces a hardware architecture based on the MBVH datastructure for major further improvements in area- and energy-efficiency compared to [Lee et al. 2014]. These gains are orthogonal to the approach of [Hwang et al. 2015] and could be combined with their hybrid representation.

3 System Architecture

As a baseline for evaluation, we consider a model architecture based largely on [Lee et al. 2014], which in turn builds on [Lee et al. 2013], shown in Figure 2. The main components of this architecture are *traversal units* (TRV) which handle processing for BVH inner nodes and *intersection units* (IST) which perform ray-triangle intersection tests using Wald's [2004] method. Ray data records enter the system from a shader processor, are assigned a free ray slot by a scheduler hardware, and then pass between TRV and IST units through FIFOs until traversal completes. Each ray is assigned a traversal stack from a specific TRV unit, and all its stack operations are performed by this TRV. This organization is inefficient in some ray states, e.g., upon finishing processing a leaf node, the ray has to return from IST to TRV and pass through the TRV pipeline to perform a stack pop, which may send the ray back to IST. However, the common case of repeated TRV operations is very fast.

The main focus of this work is on the TRV unit. The baseline TRV unit shown in Figure 3 is designed to process BVH nodes in a layout that, for each node, stores the AABBs and pointers of its two children. When a ray enters the TRV, it first attempts to fetch into memory the target node of the ray. It then performs intersection tests against the two child AABBs in parallel using the slabs test. Finally, depending on the results of the tests, the unit performs stack operations, e.g., pushing the pointers of hit children to the stack, and determines the new state of the ray. If both children are hit, they are traversed in a front-to-back order based on the distance values output from the slabs tests.

The rest of the baseline system is configured as follows. 4 TRVs are allocated per IST. Each TRV has storage for 32 stacks, therefore, up to $4 \times 32 = 128$ rays may be active simultaneously. There are two 128KB caches: a node cache shared by the TRVs and a primitive cache serving the IST. In addition, each TRV has a small 16KB L1 node cache. All caches are set as non-blocking, 4-way set-associative. The node caches use node-sized lines (64B for BVH, 128B for MBVH2), while the primitive L1 is a two-bank interleaved cache [Alvarez et al. 2007] with 64B lines per bank, to accommodate fully pipelined unaligned accesses to 48B primitives. The sizes of FIFOs are determined empirically so as to prevent deadlocks: they are first sized at 32 elements and simulated, and underutilized FIFOs are shrunk to the first power-of-two above the maximum population encountered in simulation. As a latency hiding mechanism, we use "looping for next chance" from RayCore [Nah et al. 2014]: If a memory access misses, the corresponding ray continues through the pipeline, but its subsequent computations are invalidated, and the ray is rescheduled for later execution.

We also diverge from [Lee et al. 2013] by storing full stacks on-chip instead of short stacks. As a 64-entry stack was sufficient to render all evaluated scenes by a large margin - at worst path tracing in *Hairball* used 28 entries - and only accounts for 5..10% of the area and power of the architecture. It appears that short-stack methods are more interesting for architectures with many GPU-like slow threads, or rendering algorithms that require large stack entries such as [Vaidyanathan et al. 2016].

3.1 Proposed MBVH architecture

MBVHs are structured similarly to the BVH layout in [Lee et al. 2014], except there is space for more than two children per node. In this study, we redesign the TRV unit to handle 4-wide MBVHs as shown in Figure 3. It is straightforward to increase the number of box test units to 4, and double the cache line and read port sizes in the node cache hierarchy, keeping the cache capacities constant.

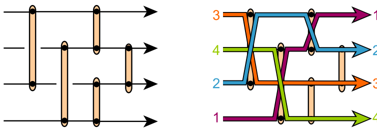


Figure 4: *Left side: Magnification of the sorting network shown in Figure 3 with 5 comparators. Right side: Example sort.*

The main new complication introduced with MBVH is that multiple child nodes may be intersected and pushed to the stack per visit: for a 4-wide MBVH, four children may be hit per TRV processing. As the cache top is kept in the ready state record, up to 3 entries may be inserted into the stack memory. This may be implemented by means of a multi-bank memory, as consecutive entries are guaranteed to be on separate banks. Moreover, it is desirable to traverse BVH nodes in a closest-first order, i.e., the references inserted to the stack should be sorted according to their distance from the camera. In BVH, this is accomplished with a simple compare-swap of the two children, but for high branching factors a sorting operation is necessary. Small sets of numbers can be easily sorted in hardware by means of *sorting networks*, where the array is passed through a series of comparators as shown in Figure 4. Knuth [Knuth 1999] gives depth-optimal networks for up to 16 inputs. For the case of four inputs, an optimal network consists of five comparators, with a *depth* of 3, i.e., the data passes through at most 3 sequential compare-swaps. We pessimistically allocate one pipeline stage per sorting network layer as shown in Figure 3, resulting in 2 extra cycles of latency compared to a BVH TRV. For occlusion rays this step might be bypassed, reducing the TRV latency. A sorting network structure has been used in software by, e.g., Guthe [Guthe 2014] to avoid branches.

Ernst and Greiner [Ernst and Greiner 2008] recommend storing the distance value of each intersected node in the stack, so that after finding a triangle intersection closer than the stored nodes, they can be rejected without testing their children. In initial testing, this technique showed significant performance gains of ca. 10% for MBVH, but only 3% for BVH. The effect is mainly due to triangle intersections avoided by the distance test. As shown in Figure 2, we equip the MBVH TRV with a distance stack, in effect doubling the stack size, while keeping the BVH in the original configuration. MBVH is typically implemented with a power-of-two branching factor to take advantage of SIMD instructions. In custom hardware, it is interesting to use other factors such as 3 or 5. The main complication of odd branching factors is memory hierarchy design: the cache hierarchy should be able to supply one node per cycle to the TRV unit, though they are unaligned in memory. Two approaches are apparent. Firsly, the node L1 may be implemented as a two-bank interleaved cache, as we do with the primitive L1. Secondly, the node cache hierarchy might be addressed with array indices rather than byte addresses, and use a node-sized data word. We experimented with MBVH3, MBVH5 and MBVH6 using both techniques, but these were sightly less efficient than MBVH4.

3.2 Evaluation

In order to evaluate the performance impact of MBVH, we implemented a cycle-level simulator for the baseline and proposed architectures. Though this paper focuses on the TRV pipeline component, the full system including the cache hierarchy and memory needs to be simulated to determine the performance effects. The simulator is split into two parts:A software ray tracer draws scenes and generates, for each ray, logs of node visits and stack operations in a compact binary format, which are then fed to an architecture simulator. The simulator models the cycle-level behavior of the components in Figure 2, including the cache hierarchy, traversal and intersection units, and the interconnection FIFOs and arbiters. The assumed clock rate is 500MHz. The main memory is modeled with Ramulator [Kim et al. 2016]. We assume a LPDDR3-1600 memory with two 32-bit channels, for a peak theoretical data rate of 12.8GB/s.

The area of each architecture is coarsely estimated by counting the number of floating-point units and memory blocks, including caches, stacks and FIFOs: we assume that e.g. control logic, clock trees and pipeline registers add a similar margin to both configurations. The simulator keeps track of activity rates for each components: when idle, they are assumed to be clock gated and contribute only static power. Dynamic and static power figures for SRAMs are obtained from CACTI 6.5 [Muralimanohar et al. 2009], at 45nm. Note that *sequential* caches are used, which first access the tag array before reading data. In the *normal* configuration, the cache reads all words in the target set simultaneously with the tag, but due to the wide read ports in this work, this produces very large caches. For FPUs we use the energy and area per FLOP figures of Galal and Horowitz [2011]. The IST division is implemented with 11 FLOPs as in the algorithm by Markstein [Markstein 2004]. DRAM power figures are produced with DRAMPower [Chandrasekar et al. 2012].

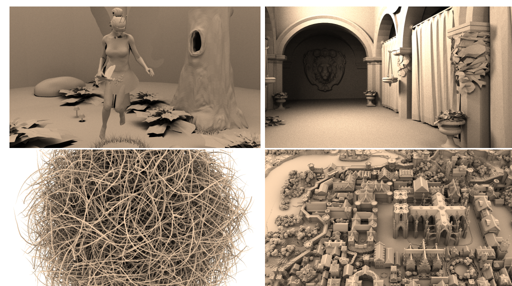


Figure 5: *Test scenes used in simulation: Fairy (179K tri.), Crytek Sponza (262K tri.), Hairball (2.9M tri.) and Rungholt (6.7M tri).*

Four test scenes (Figure 5) were rendered at a 1280x720 resolution with primary rays, as well as diffuse path tracing limited to four bounces and one sample per pixel. Secondary rays are fed to the processor when the preceding ray is complete. BVH trees are constructed with a binned SAH sweep, and MBVHs with Wald's top-down recursive splitting algorithm [Wald et al. 2008]. The latter is chosen since it appears straightforward to implement in hardware by adding extra bookkeeping to the builder unit by Doyle [2013]. Simulation results are shown in Table 2. Since a speedup is expected due to the added hardware resources in MBVH, we use size-independent figures of merit: memory traffic, energy per frame, and performance per area. Area and power breakdown for the *Fairy* scene are shown in Table 1. MBVH4 gives significant improvements in energy and area efficiency across different scenes and ray types. With diffuse rays, the improvement is driven by more efficient use of external memory. DRAM accounts for an average of 70% of energy. In addition to reducing memory traffic, the larger refill increment in MBVH utilizes DRAM better than the baseline. With primary rays, DRAM is insignificant. The efficiency gains from MBVH are smaller and more inconsistent, depending on the ratio of AABB to triangle tests: MBVH performs well in *Rungholt* and poorly in *Hairball*, extreme examples of box-heavy and triangle-heavy scenes, respectively.

3.3 Conclusion

This paper proposed to use MBVH in fixed-function ray tracing accelerators and discussed implementation techniques. 4-wide MBVH improved energy per frame by an average of 24% and performance per area by 19% with incoherent rays: therefore, it appears to be a significant low-hanging fruit in ray tracing hardware design. As future work, we are interested in whether the improvements from MBVH are cumulative with other memory-conserving techniques such as quantized trees and treelet scheduling.

Scene	Ray type	BVH				MBVH4			
		Perf. (MRPS)	Energy (mJ/frame)	Perf. / A (MRPS/mm^2)	DRAM traffic (MB)	Perf. (MRPS)	Energy (mJ/frame)	Perf. / A (MRPS/mm^2)	DRAM traffic (MB)
Fairy	primary	47	9.3	12.2	8	76 (+61%)	8.4 (-10%)	13.8 (+13%)	6 (-19%)
	diffuse	36	84	9.3	644	60 (+67%)	67 (-20%)	11.0 (+17%)	497 (-23%)
Crytek	primary	25	18	6.5	5	47 (+87%)	15 (-15%)	8.5 (+31%)	3 (-42%)
	diffuse	11	417	2.8	3332	19 (+78%)	307 (-26%)	3.5 (+25%)	2345(-30%)
Hairball	primary	14	29	3.7	80	19 (+33%)	24 (-17%)	**3.4 (-6%)**	36 (-55%)
	diffuse	6	490	1.5	4370	10 (+67%)	341 (-31%)	1.8 (+17%))	2933 (-33%)
Rungholt	primary	43	9.1	11.4	3	88 (+100%)	7.5 (-18%)	16.1 (+41%)	3 (-18%)
	diffuse	29	92	7.6	703	48 (+66%)	76 (-17%)	8.8 (+16%)	582 (-17%)
Mean	primary	-	-	-	-	+70%	-15%	+20%	-33%
	diffuse	-	-	-	-	+69%	-24%	+19%	-26%

Table 2: *Comparison of BVH and MBVH4 accelerators. MBVH4 is consistently more efficient except for primary rays in Hairball, where IST is a bottleneck. With incoherent rays, DRAM dominates energy consumption.*

Unit	TRV	IST	stack	cache	fifo	dram	$\sum$
BVH							
Area (mm^2)	0.25	0.20	0.25	1.57	1.25	-	3.51
P. power (mW)	237	74	13	90	23	37	475
D. power (mW)	157	60	10	130	25	608	990
MBVH4							
Area (mm^2)	0.41	0.20	0.51	2.35	1.25	-	5.47
P. power (mW)	413	88	22	101	22	42	689
D. power (mW)	285	71	18	153	24	767	1319

Table 1: *Area estimate for baseline (BVH) and proposed (MBVH), and power breakdown on Fairy, primary and diffuse rays. Caches have the same capacity in MBVH but take up more area and power due to the wider read port. The DRAM is a clear bottleneck for incoherent rays.*

Acknowledgement

The authors would like to thank Finnish Funding Agency for Technology and Innovation (project Parallel Acceleration 3, funding decision 1134/31/2015) and European Commission in the context of ARTEMIS project ALMARVI (ARTEMIS 2013 GA 621439)", as well as the TUT graduate school and the Nokia Foundation. Models used are courtesy of Ingo Wald (Fairy), Frank Meinl (Crytek Sponza), Samuli Laine (Hairball) and kescha (Rungholt).

References

ALVAREZ, M., SALAMI, E., RAMIREZ, A., AND VALERO, M. 2007. Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *IEEE Int. Symp. Performance Analysis of Systems Software*, 62–71.

CHANDRASEKAR, K., WEIS, C., LI, Y., AKESSON, B., WEHN, N., AND GOOSSENS, K., 2012. DRAMPower: Open-source DRAM power & energy estimation tool. http://www.drampower.info.

DAMMERTZ, H., HANIKA, J., AND KELLER, A. 2008. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Comput. Graph. Forum 27*, 4, 1225–1233.

DOYLE, M. J., FOWLER, C., AND MANZKE, M. 2013. A hardware unit for fast SAH-optimised BVH construction. *ACM Trans. Graph. 32*, 4, 139.

ERNST, M., AND GREINER, G. 2008. Multi bounding volume hierarchies. In *IEEE Symp. Interactive Ray Tracing*, 35–40.

GALAL, S., AND HOROWITZ, M. 2011. Energy-efficient floating-point unit design. *IEEE Trans. Comp. 60*, 7, 913–922.

GUTHE, M. 2014. Latency considerations of depth-first GPU ray tracing. In *Eurographics (Short Papers)*, 53–56.

HWANG, S. J., LEE, J., SHIN, Y., LEE, W.-J., AND RYU, S. 2015. A mobile ray tracing engine with hybrid number representations. In *SIGGRAPH Asia Mobile Graph. Interact. Appl.*, 3.

KIM, Y., YANG, W., AND MUTLU, O. 2016. Ramulator: A fast and extensible DRAM simulator. *IEEE Comp. Arch. Letters 15*, 1 (Jan), 45–49.

KNUTH, D. E. 1999. *The Art of Computer Programming: Volume 3: Sorting and Searching*, vol. 3.

LEE, W., SHIN, Y., LEE, J., KIM, J., NAH, J., JUNG, S., LEE, S., PARK, H., AND HAN, T. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. High-Performance Graph.*, 109–119.

LEE, J., LEE, W.-J., SHIN, Y., HWANG, S., RYU, S., AND KIM, J. 2014. Two-AABB traversal for mobile real-time ray tracing. In *SIGGRAPH Asia Mobile Graph. Interact. Appl.*, 14.

MARKSTEIN, P. 2004. Software division and square root using Goldschmidt's algorithms. In *Proc. Conf. Real Numbers and Comp.*, vol. 123, 146–157.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories*, 22–31.

NAH, J.-H., KWON, H.-J., KIM, D.-S., JEONG, C.-H., PARK, J., HAN, T.-D., MANOCHA, D., AND PARK, W.-C. 2014. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph. 33*, 5, 162.

VAIDYANATHAN, K., AKENINE-MÖLLER, T., AND SALVI, M. 2016. Watertight ray traversal with reduced precision. In *Proc. High-Performance Graph-*, Eurographics Association, 33–40.

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *IEEE Symp. Interact. Ray Tracing*, 49–57.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, Germany.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3, 434–444.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-4151-3
ISSN 1459-2045