

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING
DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION



MASTER'S THESIS

Wide Bounding Volume Hierarchies for Ray Tracing

Bc. Lukáš Cezner

Supervisor: doc. Ing. Jiří Bittner Ph.D.

Study Program: Open Informatics

Specialization: Computer Graphics

May 2025

I. Personal and study details

Student's name: **Cezner Lukáš** Personal ID number: **498875**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Graphics and Interaction**
Study program: **Open Informatics**
Specialisation: **Computer Graphics**

II. Master's thesis details

Master's thesis title in English:

Wide Bounding Volume Hierarchies for Ray Tracing

Master's thesis title in Czech:

Široké hierarchie obalových těles pro metody sledování paprsku

Name and workplace of master's thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **07.02.2025**

Deadline for master's thesis submission: _____

Assignment valid until: **20.09.2026**

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean's signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

I. Personal and study details

Student's name: **Cezner Lukáš** Personal ID number: **498875**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Graphics and Interaction**
Study program: **Open Informatics**
Specialisation: **Computer Graphics**

II. Master's thesis details

Master's thesis title in English:

Wide Bounding Volume Hierarchies for Ray Tracing

Master's thesis title in Czech:

Široké hierarchie obalových těles pro metody sledování paprsku

Guidelines:

Review methods using the bounding volume hierarchy (BVH) to accelerate ray tracing. Focus on techniques for building and traversing so-called wide BVHs, i.e., hierarchies with higher branching factors. Implement a suitable method for constructing such a BVH and the associated traversal algorithm. Implement also a selected compression scheme for BVH nodes. Optionally exploit view dependency, i.e., optimize the wide BVH for a particular camera position in the scene. Integrate the implementation into an existing project that uses the Vulkan API. Based on detailed profiling, propose optimizations for the implementation that will maximize the rendering speed for recent GPU architectures. Perform thorough testing of the BVH construction and rendering speeds using path tracing in at least five test scenes.

Bibliography / sources:

- [1] Gu, Y., He, Y., Blueloch, G. E. (2015). Ray specialized contraction on bounding volume hierarchies. Computer Graphics Forum, 34(7), 309-318.
- [2] Ylitie, H., Karras, T., Laine, S. (2017). Efficient incoherent ray traversal on GPUs through compressed wide BVHs. Proceedings of High Performance Graphics, 1-13.
- [3] Ogaki, Shinji, Derouet-Jourdan, A. (2016). An N-ary BVH child node sorting technique for occlusion tests. Journal of Computer Graphics Techniques, 5(2).
- [4] Benthin, C., Meister, D., Barczak, J., Mehalwal, R., Tsakok, J., Kensler, A. (2024). H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 7(3), 1-14.
- [5] Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., Bittner, J. (2021). A survey on bounding volume hierarchies for ray tracing. In Computer Graphics Forum, 40(2), 683-712.

DECLARATION

I, the undersigned

Student's surname, given name(s): Cezner Lukáš
Personal number: 498875
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

Wide Bounding Volume Hierarchies for Ray Tracing

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc. Lukáš Cezner

.....
student's signature

Acknowledgements

I would like to thank my thesis supervisor doc. Ing. Jiří Bittner Ph.D. for his guidance and valuable advice. I would also like to thank my family and friends for their support during my studies.

Abstract

Path tracing, a technique for generating photorealistic images, is widely used in offline rendering and is increasingly adopted in real-time applications. Modern ray tracing frameworks typically utilize wide bounding volume hierarchies (BVH), i.e., hierarchies with a branching factor greater than two. This thesis investigates methods for constructing and traversing such acceleration structures. We implemented wide BVH construction using two heuristics (SAH and view-dependent), two memory layouts (uncompressed and quantized bounding volumes), and evaluated their traversal using two shading languages (GLSL and Slang).

Across five different scenes, the evaluation showed an average speedup in secondary rays of 20% for 4-ary uncompressed BVHs and 78% for quantized 6-ary BVHs compared to binary BVHs. View-dependent BVHs demonstrated an approximate performance gain of +9% for primary rays and +4% for secondary rays in 4-ary BVH. In uncompressed BVHs, Slang shaders achieved a modest speedup of 1% to 2% for secondary rays compared to GLSL, whereas in quantized BVHs, they exhibited reduced performance in range from -9% to -14%.

Keywords: path tracing, wide BVH, view-dependent, compressed BVH, Slang

Abstrakt

Sledování paprsků, jako technika pro generování fotorealistických obrazů, se široce používá při offline vykreslování a stále více se prosazuje v aplikacích pracujících v reálném čase. Moderní nástroje pro sledování paprsků obvykle využívají široké hierarchie obalových těles (BVH), tj. hierarchie s faktorem větvení větším než dva. Tato práce zkoumá metody konstrukce a procházení takových akceleračních struktur. Implementovali jsme konstrukci širokých BVH pomocí dvou heuristik (SAH a závislé na pohledu), dvou paměťových rozložení (nekomprimované a s kvantizovanými obalovými tělesy) a vyhodnotili jsme jejich traverzaci pomocí dvou programovacích jazyků pro shadery (GLSL a Slang).

V pěti různých scénách bylo změřeno průměrné zrychlení sekundárních paprsků o 20 % u 4-árních nekomprimovaných BVH a o 78 % u kvantizovaných 6-árních BVH ve srovnání s binárními BVH. BVH závislé na pohledu vykazovaly přibližný nárůst výkonu o +9 % pro primární paprsky a +4 % pro sekundární paprsky u 4-árních BVH. U nekomprimovaných BVH dosáhly Slang shadery mírného zrychlení o 1 % až 2 % pro sekundární paprsky oproti GLSL, zatímco u kvantizovaných BVH vykazovaly snížení výkonu v rozsahu -9 % až -14 %.

Klíčová slova: sledování paprsků, široké BVH, pohledová závislost, komprimované BVH, Slang

Contents

Contents	IX
1 Introduction	1
1.1 Whitted Ray Tracing and Path Tracing	1
1.2 Bounding Volume Hierarchy (BVH)	2
1.2.1 Bounding Volumes	2
1.2.2 Surface Area Heuristic (SAH)	3
1.3 GPU Architecture	3
2 Related Work	5
2.1 Construction Methods	5
2.1.1 Binary BVH Construction	5
2.1.2 Contraction Based on Reducing Pass Tests	6
2.1.3 Contraction Based on Dynamic Programming	7
2.1.4 HPLOC	8
2.2 Tree Compression	9
2.3 Traversal Methods	10
2.3.1 Thread Divergence	11
2.3.2 Persistent Threads	11
2.3.3 Speculative Traversal	11
2.3.4 Traversal Order of Children	11
3 Shading Languages	15
3.1 GLSL	15
3.2 Slang	15
4 Implementation	17
4.1 Orchard	17
4.2 Wide BVH Construction	18
4.2.1 Surface Area Contraction	18
4.2.2 View-dependent Contraction	18
4.3 Wide BVH Memory Layout	18
4.3.1 Uncompressed Wide BVH Layout	19
4.3.2 Quantized Wide BVH Layout	20
4.4 Wide BVH Traversal	21
4.5 Support Tools	22
4.5.1 nvidia-stabilize	22
4.5.2 compare-benchmark	23
4.5.3 slang-depbuild	23
5 Difficulties During Implementation	25

5.1	Shader Resource Allocation	25
5.1.1	In Single Kernel Traversal	26
5.1.2	In Quantized Wide BVH Traversal	28
5.2	Instruction Hoisting in Slang	29
5.3	Subgroup Coherency in Slang	31
6	Results	33
6.1	Dataset	33
6.2	Testing Environment	34
6.2.1	Workgroup Size	34
6.3	SAH Traversal Cost	35
6.4	CBTC Heuristic Threshold	36
6.5	Depth Limit for Ray Statistics	36
6.6	Number of Sample Rays	37
6.7	Overall Results	38
7	Conclusion and Future Work	41
	References	43
	List of Figures	47
	List of Tables	48
	List of Listings	48
	Appendices	49
A	MRps Performance for Various Workgroup Configuration	49
B	Per-Scene Results	52

1 Introduction

Commonly used methods for producing realistic computer-generated images include ray tracing and path tracing. The illumination in the scene is simulated by casting numerous light rays that bounce off objects according to probabilities determined by the material properties of those objects^[1]. A huge number of rays must be cast to produce a high-quality image, which is highly computationally demanding. To speed up this process, a frequently utilized acceleration structure is the bounding volume hierarchy (BVH; see Section 1.2).

Rendering a scene consists of two steps: constructing a BVH for the scene and traversing this BVH to find ray-primitive intersections. There are many types of BVHs and diverse methods for building them. Notably, BVHs can be categorized into two classes: simple binary trees and those with a higher branching factor (node arity), commonly referred to as wide BVHs. The advantage of wide BVHs is a lower tree depth and fewer nodes, resulting in reduced memory consumption and potentially increased performance^[2].

This thesis examines wide BVHs, explores methods for their construction, and compares their traversal to that of binary BVHs. We outline our implementation of wide BVH construction using two distinct heuristics (SAH and view-dependent) and two memory layouts (uncompressed and quantized), and our traversal shader implemented in two shading languages (GLSL and Slang). The challenges faced during the development process are reviewed in Section 5. We evaluated the implemented wide BVHs against a binary BVH across five scenes and analyzed various parameter configurations (shader workgroup size, traversal cost, view-dependent heuristic threshold, and parameters of sample rays) to achieve optimal performance.

1.1 Whitted Ray Tracing and Path Tracing

There are two methods that are often mistaken for each other: Whitted Ray Tracing^[3] and Path Tracing^[1]. Both techniques employ primary rays (originating from the camera), secondary rays (reflected from materials), and shadow rays (which determine whether a light source is occluded).

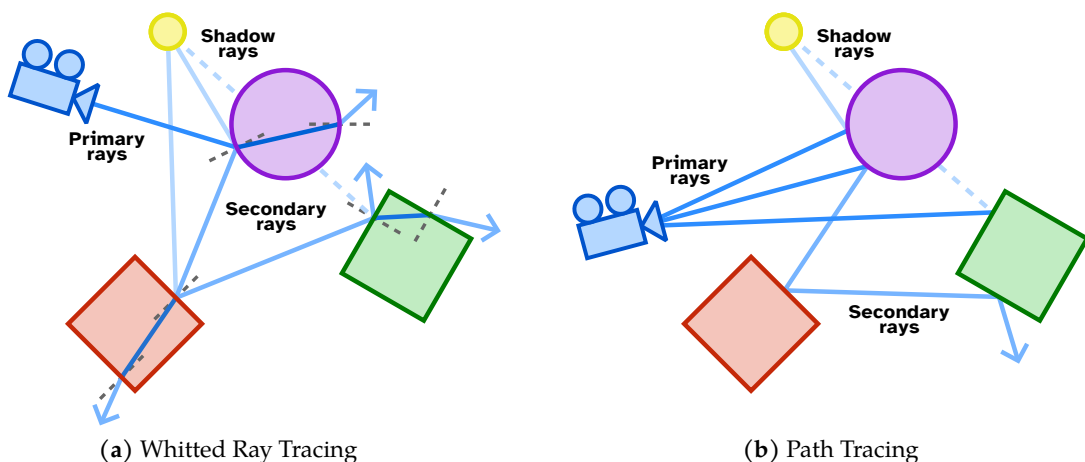


Figure 1: Schematic visualization of Whitted Ray Tracing and Path Tracing.

Whitted Ray Tracing^[3] simulates illumination through perfect reflections, refractions, and diffusion. Each ray that hits an object is split into more rays: a reflection ray that bounces according to the law of reflection (reflection angle equals incidence angle), a refraction ray that obeys Snell’s law (if the material has refractive properties), and a shadow ray for each light source. These rays are recursively cast until terminated (by a depth limit or if they do

not intersect)^[3]. Therefore, Whitted Ray Tracing has several shortcomings: it cannot simulate complex light properties and generates rays that rapidly increase with increased depth.

Path tracing was designed as a method for numerical Monte Carlo integration of the rendering equation^[1]. Instead of branching the ray into more rays after each bounce, only a single ray is cast in a direction determined by random probability, usually using importance sampling (prioritizing directions with high contribution). Shadow rays can be cast from each ray bounce to calculate direct illumination. This technique produces high-frequency noise, requiring multiple rays (samples) per pixel to mitigate the noise^[1].

1.2 Bounding Volume Hierarchy (BVH)

The bounding volume hierarchy (BVH) is a rooted tree built upon scene primitives (usually triangles). Each leaf node contains one or more primitives, and each internal node consists of a bounding volume: a closed region that completely contains all primitives from the entire subtree^[4]. BVH is an object hierarchy. Each internal node splits a set of primitives into disjoint subsets, one per child. Although children's bounding volumes may overlap, unlike spatial subdivisions, which divide a node's space into disjoint subspaces, this method provides a predictable memory footprint, since each primitive is stored only once in the tree^[2]. Figure 2 shows an example of a simple BVH tree.

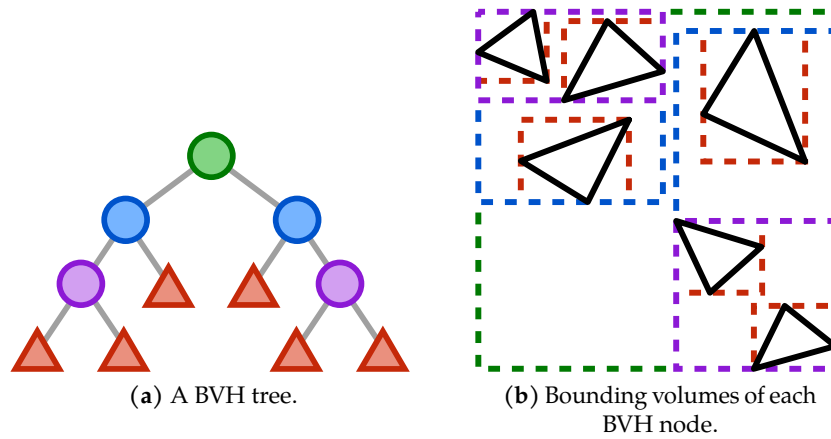


Figure 2: An example of a BVH tree. The color of internal nodes indicates tree depth: the root is green, depth 1 nodes are blue, and depth 2 nodes are purple. Leaf nodes are marked in red.

Detection of the nearest intersection between an input ray and primitives in the BVH is achieved through depth-first search (DFS) traversal using a traversal stack. Starting from the root node, an intersection between the node's bounding volume and the ray is computed. If one is found, the child nodes are pushed onto the stack. When the traversal reaches a leaf, the intersection between the leaf primitives and the ray is calculated, and the closest intersected primitive encountered so far is saved. The distance to the nearest primitive can be used for tree pruning, allowing nodes beyond this distance to be skipped^[2].

1.2.1 Bounding Volumes

BVH can utilize a variety of bounding volume types. Effective bounding volumes should primarily possess properties such as inexpensive ray-intersection tests, tight fitting, and small memory requirements, but these properties are always subject to trade-offs among each other. When considering memory requirements and the cost of ray-intersection tests, the bounding sphere is the most cost-effective option, though it does not offer a particularly precise boundary. On the opposite end of the spectrum are convex hulls, which are used exclusively for collision detection^[5]. Typically, for path tracing, an axis-aligned bounding box (AABB) is used, though

other bounding volumes are also a subject of research. Figure 3 illustrates some of the variants of bounding volumes.

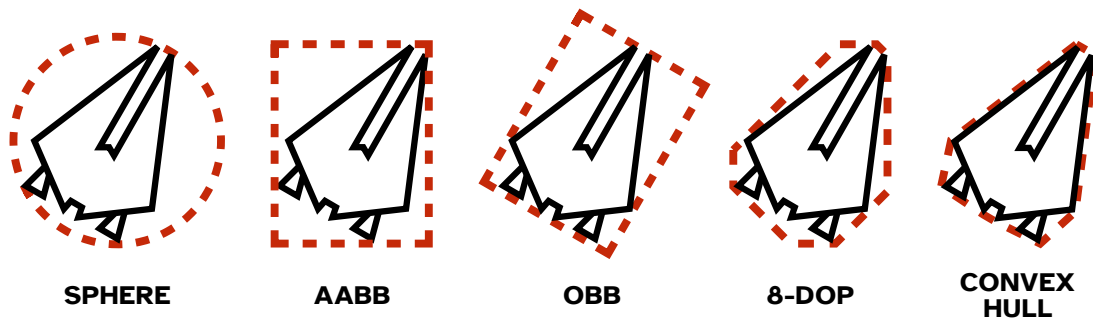


Figure 3: Usual types of bounding volumes: bounding sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull. Adapted from an image by Ericson^[5].

1.2.2 Surface Area Heuristic (SAH)

The way in which primitives in a BVH are divided is essential for the quality of the resulting BVH. To evaluate BVH quality, a heuristic to estimate traversal cost is widely used. The most commonly used one is the Surface Area Heuristic (SAH), which approximates the conditional probability of traversing a node, given that its parent was hit, by the ratio between the surface areas of the child and the parent^[6]. Subsequently, the cost of subtree of a node N can be formulated as:

$$C(N) = c_t \cdot \sum_{N_{internal}} \frac{SA(N_{internal})}{SA(N)} + c_i \cdot \sum_{N_{leaf}} \frac{SA(N_{leaf})}{SA(N)},$$

where $SA(N)$ is the surface area of a node N , $N_{internal}$ and N_{leaf} are the internal and leaf nodes of a subtree, and c_t and c_i are the costs of traversing a node and computing a ray-primitive intersection.

The SAH does not accurately capture every aspect of ray tracing behavior, as it is based on several assumptions that may not hold true, specifically that rays are uniformly distributed over the scene and that they are not occluded by primitives^[6].

1.3 GPU Architecture

In order to achieve high parallelization of shader execution, a GPU is organized into a hierarchy. At the bottom, a warp (in NVIDIA terminology) is composed of 32 (NVIDIA and AMD) or 64 (AMD)^[7] parallel threads, which are scheduled together and execute only one common instruction at a time. When threads within a warp diverge due to a conditional branch, the warp processes only one branch at a time, deactivating threads that are not on the current branch path^[8].

Warps are bundled into a compute unit (called a streaming multiprocessor in NVIDIA terminology). Each compute unit contains an L1 cache, which is partially used as shared memory, a low-latency memory with a limited access scope. A compute unit maintains execution contexts for more warps than are physically present in hardware, facilitating thread-level parallelism to hide latencies. These warps are called active warps. The warp scheduler selects an instruction ready for execution, either from the same warp as before if there are no unresolved dependencies, or from a different active warp^[8].

Vulkan defines an abstract execution model based on the GPU hierarchical architecture. The compute shader is divided into a 3D grid of workgroups, which are consecutively assigned

to individual compute units. Within a workgroup, shared memory can be allocated, which is visible only to that workgroup. Each workgroup is composed of a 3D array of invocations, with each invocation representing a single thread in execution. Dividing computation into a 3D grid simplifies the mapping of multidimensional data computations^[9]. Figure 4 illustrates the hierarchical structure of the execution model.

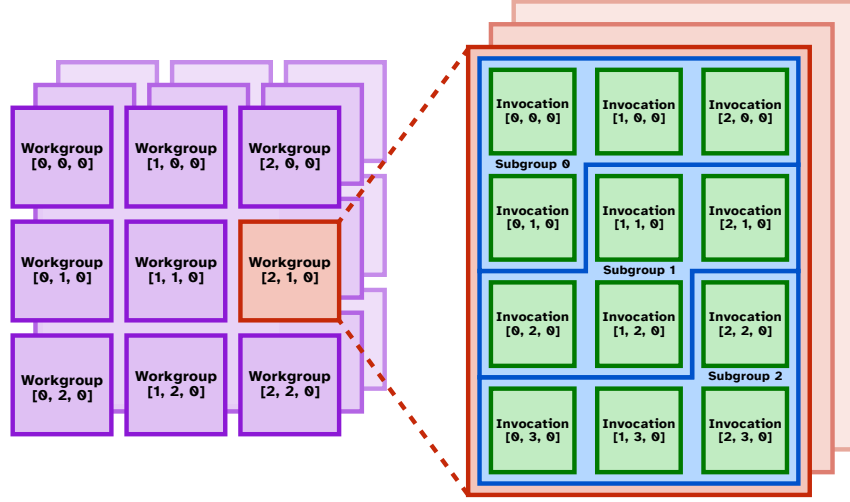


Figure 4: Example of the Vulkan execution model. The relationship between workgroups, subgroups (4 invocations in this case), and shader invocations.

Shader invocations within one workgroup are also partitioned into subgroups, defined as a group in which invocations can synchronize and share data with each other^[10]. As the Vulkan execution model is a logical abstraction over the hardware, it is not explicitly stated that a subgroup usually corresponds to a warp, and data sharing between invocations inside a subgroup is done using warp communication intrinsics. Therefore, in this thesis, we configure the workgroup sizes as multiples of subgroups (warps) to avoid the creation of unused threads within warps.

2 Related Work

This section provides an overview of current techniques for BVH construction, compression, and methods for efficient BVH traversal on the GPU, with particular focus on wide BVHs.

2.1 Construction Methods

The vast majority of algorithms for constructing a wide BVH are based on an existing source binary BVH and its transformation. These algorithms typically perform a sequence of contractions, in which an internal node is selected for removal from the tree and replaced by its child nodes. Each contraction increases the arity of the parent node, leading to the creation of a wide BVH node, as shown in Figure 5.

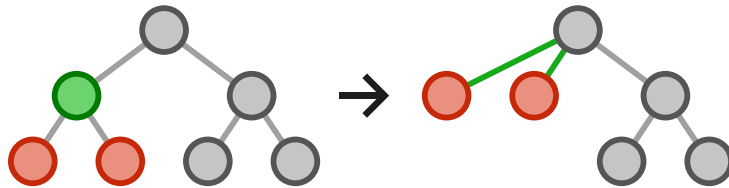


Figure 5: Example of the contraction of an internal BVH node. Initially, the green node is contracted, resulting in its children being moved to the root, which consequently increases the arity of the root.

2.1.1 Binary BVH Construction

Before discussing wide BVH contraction techniques, it is useful to briefly summarize binary BVH contraction methods. As stated in the Introduction, the methods for building them are very diverse. They can be classified into three categories: top-down construction, which starts from the root containing all primitives and iteratively splits them; bottom-up construction, which begins with individual primitives as leaves and progressively merges them; and topology optimization, which modifies an existing BVH to enhance its quality. For each category, we discuss an existing method suitable for GPUs.

Linear BVH (LBVH)

LBVH^[11] utilizes the Morton code, a space-filling curve that divides space into a uniform grid with distinct cell indices, to index each primitive and build a radix tree on these indexed primitives. Subsequently, a BVH tree is derived from the radix tree by calculating bounding volumes for each node. Lauterbach et al.^[11] formulated this approach as a top-down construction, with a kernel launch for each tree level, but Apetrei^[12] demonstrated that the entire construction can be performed in linear time with a single kernel launch.

Parallel Locally-Ordered Clustering (PLOC)

PLOC^[13] is a bottom-up construction method that also uses the Morton code, but in a different way. Clusters, consisting of primitives or internal BVH nodes, are sorted along the Morton curve. Each cluster then searches for its nearest neighbor within a limited index range from its position in the sorted array. If the nearest neighbor relation is mutual, the pair of clusters is merged, creating a new BVH parent node. This process is performed iteratively until the entire BVH tree is constructed.

Recently, this approach was simplified by Benthin et al., reducing the number of kernel launches required per iteration, initially from five to three^[14], and later to a single kernel execution for the entire BVH construction^[15].

Treelet Restructuring (TRBVH)

TRBVH^[16] is one of the topology optimization techniques. Treelets, which are small fixed-size subtrees of an existing BVH, are restructured using dynamic programming. Starting from

the bottom, treelets are formed from a node and a fixed number of its descendants (not necessarily leaf nodes), and a new optimal BVH subtree is constructed from the treelet leaves to minimize the total SAH cost. As the method progresses toward the root, it produces a high-quality BVH.

2.1.2 Contraction Based on Reducing Pass Tests

For the purpose of wide BVH construction, Gu et al.^[17] classified bounding volume tests in tree traversal into two categories: the *Pass test* and the *Prune test*. The *Pass test* indicates that there is an intersection with at least one child's bounding volume, requiring traversal of part of the internal BVH node's subtree, since a primitive intersection may exist. In contrast, the *Prune test* indicates that none of the children's bounding volumes are intersected, and the entire BVH node's subtree can be skipped.

They observed that reducing the number of pass tests leads to faster BVH traversal and formulated a cost function $\delta(N)$ as the difference between the cost $C_{contracted}$ when node N is contracted and the cost $C_{original}$ in the uncontracted case^[17]:

$$\begin{aligned}\delta(N) &= C_{contracted} - C_{original} \\ &= (n_{children} \cdot c_b) - ((\alpha_N \cdot (1 + n_{children}) + (1 - \alpha_N)) \cdot c_b) \\ &= ((1 - \alpha_N) \cdot n_{children} - 1) \cdot c_b,\end{aligned}$$

where α_N represents the probability of the *Pass test*, c_b is the cost of a bounding volume intersection test, and $n_{children}$ denotes the number of children for node N .

If this cost is negative, the contraction of the BVH node N decreases the traversal cost and it is beneficial to perform the contraction. The condition $\delta(N) < 0$ can be rewritten as^[17]:

$$\alpha_N > 1 - \frac{1}{n_{children}}$$

The series of contractions is performed in a top-down manner. The contraction process begins at the root node and proceeds by contracting a node's child that satisfies the condition, continuing this process until no child satisfies it. After that, the algorithm recursively descends to each child node down until it reaches the leaves. During the processing of a node, the number of children $n_{children}$ is not known in advance. Therefore, the authors empirically set the condition to $\alpha_N > 0.6$ ^[17].

It should be noted that this approach does not aim to generate BVH nodes with complete child occupancy (i.e., nodes with n children in an n -ary BVH tree). Instead, the number of children in a node may vary, even at the top level of the tree, depending on α_N .

The estimation of α_N depends on the chosen heuristic. Two variants have been proposed: Surface-Area Guided Tree Contraction (SATC) and Ray-Distribution Guided Tree Contraction (RDTC).

Surface-Area Guided Tree Contraction (SATC)

The SATC utilizes information on structural imbalances of the scene, analogously to the commonly used SAH^[6] (and exhibits identical conditions that must be satisfied to yield valid results), using the ratio between the surface area of the child's bounding volume and the parent's bounding volume:

$$\alpha_N = \frac{SA(N)}{SA(N_{parent})}$$

Ray-Distribution Guided Tree Contraction (RDTC)

The RDTC, in contrast to the SATC, aims to also reflect ray-distribution imbalances from a specific view of the scene. To accomplish this, it is necessary to measure statistical data by casting a set of sample rays and tracking the number of visits for each BVH node. According to their research, approximately 0.1% to 0.5% of the rays are sufficient to achieve a high-quality contraction. With the sampled data, the probability of a *Pass test* can be represented by the ratio of visits to a node N compared to its parent node N_{parent} ^[17]:

$$\alpha_N = \frac{visitCount(N)}{visitCount(N_{parent})}$$

Furthermore, using statistical data, contraction can be restricted only to the important part of the tree (i.e., the BVH nodes that are often visited) by stopping the recursion when the visit count of a BVH node N falls below the specified threshold t (i.e., $visitCount(N) < t$). Thus, the time required to construct the BVH can be reduced^[17].

Constant Branches Tree Contraction (CBTC)

For use in a wide k -ary BVH, they proposed a very similar method. However, instead of deciding whether the cost is negative and a node should be contracted, the child with the highest probability for a *Pass test* (and therefore the highest α_N) is always contracted, until the number of children of a parent node reaches k .

Across most of the tree, the RDTC formula is used to calculate the probability α_N , whereas at the lower levels of the BVH tree, where the ray sample count is less than the threshold t , the SATC formula is applied^[17]:

$$\alpha_N = \begin{cases} \frac{visitCount(N)}{visitCount(N_{parent})} & \text{if } visitCount(N) > t, \\ \frac{SA(N)}{SA(N_{parent})} & \text{otherwise.} \end{cases}$$

2.1.3 Contraction Based on Dynamic Programming

Ylitie et al.^[18] described the construction of a wide BVH in an effort to achieve the highest possible quality without compromising time. They started with a high-quality binary BVH containing a single primitive per leaf. This binary BVH is generated by an offline CPU algorithm. The wide BVH transformation simultaneously optimizes both the internal and leaf nodes with respect to the total SAH cost^[18].

During the first pass, the optimal SAH cost $C(N, i)$ is computed for each binary BVH node. This cost reflects the total SAH cost for the entire subtree of node N , represented as a forest with at most i BVHs, where $i \in [1; k - 1]$ for a k -ary BVH tree. In the second pass, a wide BVH tree is constructed by backtracking the decisions from the cost computation^[18].

The cost $C(N, i)$ is computed using dynamic programming, starting from the leaves in a bottom-up manner. Each leaf is initialized as^[18]:

$$\forall i : C(N_{leaf}, i) = \frac{SA(N_{leaf})}{SA(N_{parent})} \cdot c_i.$$

The cost of an internal node is determined once the costs for all its child nodes have already been calculated, as described below. $C(N, i)$ for $i = 1$ characterizes the scenario in which the binary node N is converted into a new wide BVH node, either as a leaf node or an internal

node, whichever has a lower cost. Creating a leaf node is restricted by the maximum count of primitives P_{max} and is computed as the cost of intersecting P_N primitives^[18]:

$$C_{leaf}(N) = \begin{cases} \frac{SA(N)}{SA(N_{parent})} \cdot P_N \cdot c_i & \text{if } P_N \leq P_{max}, \\ \infty & \text{otherwise.} \end{cases}$$

Calculating the cost $C(N, 1)$ of an internal BVH node is more complicated. It requires selecting up to k children from the subtree that lead to the minimal cost, and this is where dynamic programming is used. The children of a wide BVH node are determined by combining the forests of the left child N_{left} and the right child N_{right} of the binary BVH node N . All possible combinations of forest sizes are taken into account, but only those combinations for which the sum of the sizes of the left and right forests equals k need to be tested. This is because $C(N, i)$ represents the cost of a forest with up to i trees, rather than with exactly i trees. Therefore, it can be described as^[18]:

$$C_{internal}(N) = C_{distribute}(N, k) + \frac{SA(N)}{SA(N_{parent})} \cdot c_t,$$

$$C_{distribute}(N, k) = \min_{i \in [1, k-1]} [C(N_{left}, i) + C(N_{right}, k - i)].$$

The cost function with $i \neq 1$ is computed in a similar manner. It represents a situation where the binary BVH node may or may not be contracted. In such cases, this binary node is not transformed into a wide BVH node; instead, it represents a list, essentially a forest of BVH subtrees, that may become potential children of a parent node. Depending on the resulting cost, a forest with i trees or fewer (possibly due to the recursive structure expressed as $C(N, i - 1)$) is selected. Therefore, the computation of the cost $C(N, i)$ can be defined as follows^[18]:

$$C(N, i) = \begin{cases} \min [C_{leaf}(N), C_{internal}(N)] & \text{if } i = 1, \\ \min [C_{distribute}(N, i), C(N, i - 1)] & \text{otherwise.} \end{cases}$$

2.1.4 HPLOC

Benthin et al. present the HPLOC algorithm^[15] with the primary objective of optimizing the PLOC++^[14] algorithm for constructing binary BVH on the GPU, while also describing the conversion to wide BVH. The conversion is performed in a top-down manner and computed in a single launch of the kernel. The proposed heuristic for selecting nodes to be contracted is to choose the internal node with the largest surface area, though any other heuristic can be easily applied^[15].

This method maintains a pair of indices for each shader invocation, stored in global memory: the first index refers to the binary BVH node to be processed by the invocation, and the second index indicates the memory location where the resulting wide BVH node will be stored. Initially, every pair of indices is set to an invalid state (representing an invalid index of a binary BVH node). Only the first shader invocation has a pair set to the root node of the binary BVH and the start of the output wide BVH node array^[15].

Each shader invocation atomically polls the pair of indices from global memory until it contains valid indices. Once valid, it retrieves the binary BVH node using the provided index and identifies n suitable children for the wide BVH node within the subtree, according to the selected heuristic. These n BVH nodes are allocated from the output array using an atomically incremented counter and assigned as children of the currently processed wide BVH node. The children are then scheduled for processing: the first child is assigned to the same shader

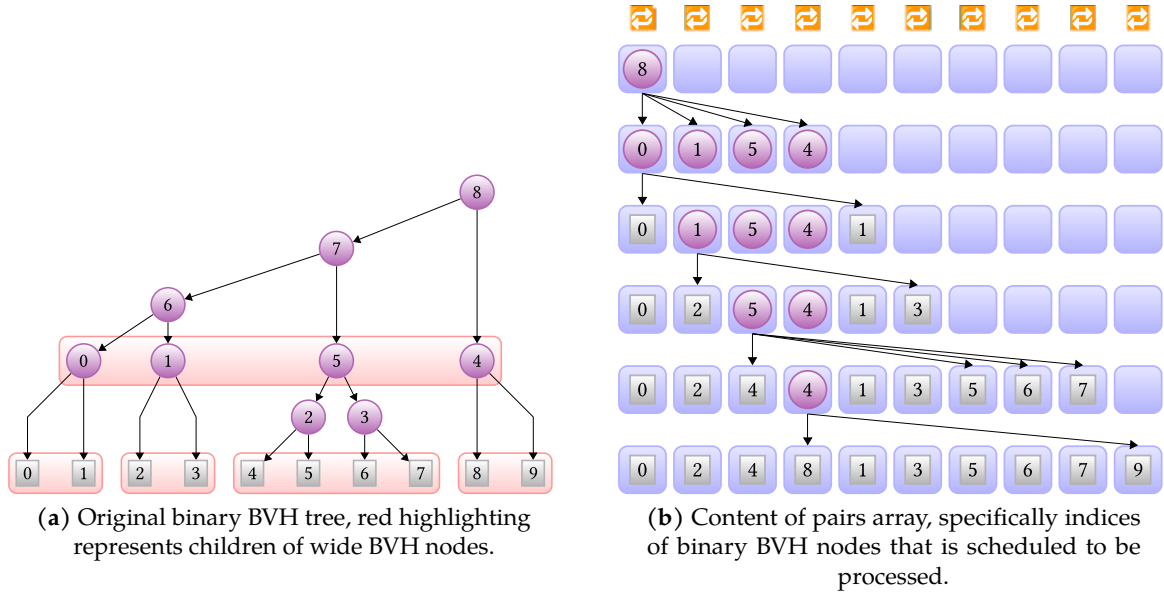


Figure 6: Example of conversion from a binary BVH tree to a 4-ary wide BVH tree using the HPLOC algorithm^[15]. Starting from the binary BVH root, the first shader invocation determines the children of the wide BVH root. It schedules itself to process the first child (internal node 0), while the remaining children (internal nodes 1, 5, and 4) are assigned to different shader invocations. In the next iteration, the first shader invocation queues a leaf, and upon processing it, the invocation terminates. Image taken from Benthin et al.^[15].

invocation, while the remaining $n - 1$ children are assigned to other invocations. This assignment is done by atomically writing each binary+wide BVH index pair (for example, using an `int64` atomic operation) into an element of the pairs array that currently holds an invalid state, which can be managed using another atomically incremented counter, since a pair with valid indices is never overwritten back to an invalid state. Finally, the created wide BVH node is written to the output array^[15].

When the processed binary BVH node is a leaf, it has no children to explore. In this case, the primitives from the binary BVH node are copied into the wide BVH node. The shader invocation then terminates, as it will not receive any more nodes to process. This means that the number of shader invocations needed to process the entire BVH is equal to the number of leaf nodes in the binary BVH^[15].

A prerequisite for this algorithm is that workgroups must be launched in increasing order. If not, forward progress cannot be guaranteed, as all active workgroups may enter a busy-wait state, waiting on a workgroup that has not been (and will not be) launched^[15].

2.2 Tree Compression

As the arity of BVH nodes increases, the node size grows proportionally. This results in increased memory traffic per node, which especially affects incoherent rays. Therefore, reducing memory traffic by compressing BVH nodes is desirable. A particularly promising method is described by Ylitie et al.^[18], who introduced a compressed 8-ary BVH. Combined with additional optimizations, such as a compressed traversal stack and an octant-based traversal order (see Section 2.3.4), this approach yields approximately a $2\times$ performance compared to other techniques^[18].

Compression is achieved by storing quantized AABBs and using bitfields efficiently. The quantization grid is shared among all child AABBs and is defined by an origin point \vec{p} , stored as 32-bit floating-point values. The grid step is expressed as exponent of powers of two (2^{e_i}) for each axis

(e_x, e_y, e_z) . Each exponent is stored as an 8-bit value extracted from a 32-bit float. The origin \vec{p} is the minimum corner \vec{b}_{lo} of the common AABB, and the grid step e_i is the smallest value such that the maximum corner \vec{b}_{hi} of the common AABB can be encoded using Q bits^[18]:

$$\vec{e} = \left\lceil \log_2 \frac{\vec{b}_{max} - \vec{p}}{2^Q - 1} \right\rceil.$$

The quantization of child AABBs must be done conservatively, ensuring that the resulting quantized AABB is not smaller than the original. This process involves rounding down the quantized result \vec{q}_{lo} of the minimum corner of the child AABB, and rounding up the maximum corner \vec{q}_{hi} ^[18]:

$$\vec{q}_{lo} = \left\lfloor \frac{\vec{b}_{lo} - \vec{p}}{2^{\vec{e}}} \right\rfloor, \quad \vec{q}_{hi} = \left\lceil \frac{\vec{b}_{hi} - \vec{p}}{2^{\vec{e}}} \right\rceil.$$

In addition to the quantization grid (\vec{p}, \vec{e}) and the AABB of the children $(\vec{q}_{lo}, \vec{q}_{hi})$, a wide BVH node must include certain metadata: indices to the node array for internal nodes, indices to the primitive array along with the number of primitives for leaf nodes, and information distinguishing which children are internal nodes and which are leaves^[18].

The internal nodes and primitives of the children are stored in contiguous memory next to each other. Therefore, a common index for all children (one for internal nodes and one for primitives) can be stored, and each child only requires an offset from this index. This offset can be stored in an 8-bit variable, along with the primitive count in the leaf node variant. The type of children is represented by an 8-bit bitfield, with each bit corresponding to a specific child^[18]. The layout of the entire BVH node is shown in Figure 7.

meta	p_x				p_y			
	p_z				e_x	e_y	e_z	$imask$
	child node base index				triangle base index			
$q_{lo,x}$	Child 0	Child 1	Child 2	Child 3	Child 4	Child 5	Child 6	Child 7
$q_{lo,y}$								
$q_{lo,z}$								
$q_{hi,x}$								
$q_{hi,y}$								
$q_{hi,z}$								

Figure 7: Compressed 8-ary BVH node^[18]. The entire node is stored in 80 bytes and contains the quantization grid (blue), quantized bounding boxes (yellow), and metadata and indexing information (green). Image taken from Ylitie et al.^[18]

During traversal, instead of decompressing bounding volumes, intersections are performed in quantization grid space by transforming the ray origin and direction from world space to this space.

2.3 Traversal Methods

The primary challenges in BVH traversal on GPUs include thread divergence, as well as the latency and bandwidth overhead associated with memory accesses. Potential solutions to these issues are described in the following sections.

2.3.1 Thread Divergence

Thread divergence can be reduced by separating the traversal of internal BVH nodes and the intersection with primitives into two distinct loops. These loops can be implemented using **if** conditions (known as if-if traversal) or **while** loops (while-while traversal)^[19]. Alternatively, a hybrid approach may be used, combining **if** and **while** constructs (i.e., if-while and while-if traversal)^[2].

```
1 ray ← fetch_ray()
2 node ← root
3 while ray is not terminated:
4     if/while node does not contain primitives:
5         traverse to the~next node
6     if/while node contains untested primitives:
7         perform a~ray-primitive intersection
```

Listing 1: Pseudocode of the BVH traversal algorithm with two separate loops: one for traversing internal BVH nodes and another for primitive intersection^[19].

This method converges threads responsible for each segment of ray traversal; however, it may struggle with a low number of active threads, particularly in the case of non-coherent rays. Two factors contribute to this behavior: long-running threads must complete their traversal of internal nodes in a **while** loop, causing delays for other threads waiting for a ray-primitive test; and the scenario where nearly all threads have terminated, leaving only a few still active in the outer loop. The first issue can be addressed by introducing an extra loop exit when the number of active threads drops below a specified threshold. The second problem can be solved using persistent threads, where threads can obtain a new ray from the work queue if their original ray has terminated^[19].

2.3.2 Persistent Threads

Persistent threads are used to bypass the hardware scheduler by implementing a software scheduler instead. A kernel is executed with a maximal launch, utilizing the maximum number of workgroups that the hardware scheduler can run concurrently. This ensures that no workgroup is waiting to be executed and that each remains active for the entire lifetime of the kernel. This approach enables cross-workgroup and device-wide synchronization on the GPU without input from the CPU. Synchronization is achieved through atomic operations in global memory, as well as mechanisms for intra-workgroup and intra-subgroup communication. As a result, it is possible to implement a software scheduler based on work queues, whether predefined at compile time or dynamically created at runtime, to allow for more fine-grained task scheduling^[20].

2.3.3 Speculative Traversal

The concept of speculative traversal proposes that rather than having inactive threads idle during the traversal of internal BVH nodes, these threads continue traversing the tree, potentially discovering (or at least approaching) a subsequent leaf to evaluate in the ray-primitive loop. The disadvantage of this approach is the additional memory bandwidth generated by these speculative threads^[19].

2.3.4 Traversal Order of Children

While determining the traversal order of intersected child nodes in a binary BVH is straightforward, the complexity increases significantly as the arity of BVH nodes increases. Some methods for solving the traversal order in wide BVHs are described in the following sections.

```

1 ray ← fetch_ray()
2 node ← root
3 leaf ← null
4 while ray is not null:
5     while node does not contain primitives:
6         traverse to the~next node
7         if node contains primitives and leaf is null:
8             leaf ← node
9             traverse to the~next node
10        if number of leafs that is not null in the~subgroup > threshold:
11            break
12    while leaf or node contains untested primitives:
13        perform a~ray-primitive intersection
14    if ray terminated:
15        ray ← fetch_ray()
16        node ← root

```

Listing 2: Pseudocode of BVH traversal while-while algorithm using persistent threads and speculative traversal with one postponed leaf^[2]. Lines 7 to 9 represent speculative traversal, lines 10 to 11 manage an early exit from the internal node traversal loop when there are enough threads for ray-primitive intersection, and lines 14 to 16 implement the persistent threads paradigm to acquire a new ray for traversal.

Sorting networks

For the first-hit traversal, the optimal order is determined by the distance of each child from the ray origin, with the closest child traversed first. Therefore, it is necessary to sort the array of intersecting children. A sorting network can be used for this purpose because it has a statically defined order of comparisons. However, in very wide BVH trees, the computational complexity of sorting the children can become significant^[2]. The optimal sorting networks for various numbers of elements are shown in Figure 8.

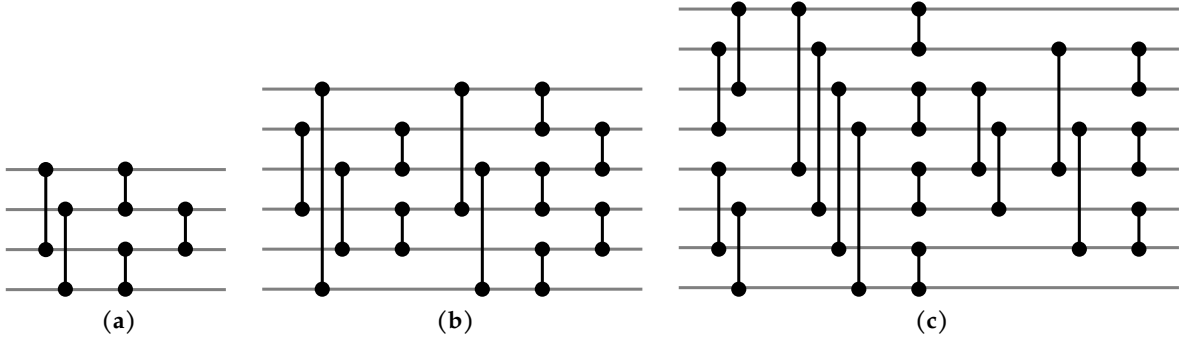


Figure 8: Optimal sorting networks for various numbers of elements: (a) 4 elements with 5 comparisons, (b) 6 elements with 12 comparisons, (c) 8 elements with 19 comparisons^[21].

Octant Based Order

Due to the computational complexity of sorting children, especially for an 8-ary BVH, Ylittie et al.^[18] generalized the traversal order originally developed by Garanzha and Loop^[22]. The children are sorted during the build stage, and the traversal order is then determined by the exclusive OR between the index i of a child and the octant r of a ray ($i \oplus r$). The octant is encoded as a binary number r , where each bit represents the sign of the ray direction vector axis. 0 means a positive sign and 1 means a negative sign^[18].

Garanzha and Loop^[22] store children in a BVH node in Morton order, based on the centroids of their bounding volumes. This approach works well only if a node has exactly 8 children

and the bounding volumes of these children are located roughly at the corners of the parent bounding volume. Therefore, Ylitie et al.^[18] optimize the order of children stored in a BVH node using the auction algorithm^[23]. They formulated $cost(N_i, s)$ as the cost of storing a child N_i at position s within the child array of a BVH node. This cost is represented by the distance of the child's bounding volume centroid \vec{c}_i from the parent's bounding volume centroid \vec{p} , projected onto the diagonal ray $\vec{d}_s = (\pm 1, \pm 1, \pm 1)$ defined by an octant s ^[18]:

$$cost(N_i, s) = (\vec{c}_i - \vec{p}) \cdot \vec{d}_s.$$

A table of size 8×8 is constructed for every pair of children N_i and position s , and the minimal total cost is determined using this table^[18].

Order for Occlusion Tests

Occlusion tests are typically used for computing shadows and resolving multi-light problems. These tests do not require determining the nearest intersection with a ray, but rather to check if any primitive exists along the ray's path (hence they are called any-hit tests). Therefore, traversing children based on their distance is not optimal. Instead, it is better to prioritize exploring the BVH subtree that has the highest probability of a ray-primitive intersection.

Ogaki and Derouet-Jourdan^[24] proposed a method for traversal order based on statistics collected from a sample of rays. They defined the cost of any-hit traversal of a BVH subtree as follows^[24]:

$$C_{any}(N) = \begin{cases} c_t + \sum_{i=1}^k \left(I_i \cdot C_{any}(N_i) \cdot \prod_{j=1}^{i-1} (1 - H_j) \right) & \text{if } N \text{ is an internal node,} \\ c_i \cdot P_N & \text{if } N \text{ is a leaf,} \end{cases}$$

where I_i represents the probability of an intersection of the child N_i , and H_j indicates the probability of a ray-primitive intersection within the subtree of the child N_j . From this equation, it can be deduced that arranging the children according to $H_i \cdot (I_i \cdot C_{any}(N_i))^{-1}$ results in minimal cost. The reordering of children is done in a bottom-up approach as part of rebuilding a BVH based on the collected ray samples. During traversal, intersected children are traversed in the same order as they are stored in a BVH node^[24].

The probabilities H_i and I_i are determined from the collected statistics by calculating the ratio of primitive hits within a child subtree (respectively, hits of the child bounding volume) to the total number of hits at the parent node. Alternatively, I_i can be approximated as $I_i = 1$, which represents the worst case where all child nodes were intersected. This approximation decreases memory consumption for storing hit statistics for each internal node but may result in suboptimal results^[24].

3 Shading Languages

As part of this thesis, we analyze the differences between two shading languages: GLSL and Slang. For use with the Vulkan API, these high-level shading languages are compiled to SPIR-V^[25], an intermediate binary language. This approach offers more flexibility in choosing a language and provides the opportunity for some offline optimization^[25].

The primary criterion for choosing a language is the performance of the resulting shaders. Specifically, there should be no overhead introduced by any language features, and the compiled output (in terms of resource usage and instructions) should be as predictable as possible. This predictability is essential for identifying optimizations in existing code. The secondary criterion, but still very important, is the possibility of minimizing redundant code caused by the presence of numerous shader variants. For different arities of a BVH tree, for various construction methods, and for each memory layout of a BVH node, there must be specific specializations of a particular shader, yet the majority of the code remains unchanged.

3.1 GLSL

The OpenGL shading language (GLSL) is the primary shading language for OpenGL and is frequently utilized for writing shaders for Vulkan as well. It is a procedural language with syntax similar to the C language. It provides fundamental programming constructs such as control statements, loops, functions, and structures, but does not provide any advanced features^[26]. Due to this, similarly to C, the limited level of abstraction offers predictable mapping to the hardware.

However, the number of ways to construct modular code is limited as well. The first option is the usage of Vulkan specialization constants. They are appropriate for determining the size of the shader workgroup and managing constant conditions in control flow, such as enabling an optional feature. However, they are not applicable for configuring the arity of a BVH tree because a specialization cannot be used as the size argument of an array in the BVH node structure. Specifically, as the GLSL specification^[26] states in Section 4.11: *"Types containing arrays sized with a specialization constant cannot be compared, assigned as aggregates, declared with an initializer, or used as an initializer."*^[26] and *"Arrays inside a block may be sized with a specialization constant, but the block will have a static layout. Changing the specialized size will not re-layout the block. In the absence of explicit offsets, the layout will be based on the default size of the array."*^[26] For instance, if the default arity of a BVH node is set to 4, resulting in a node size of 120 bytes, and we choose to specialize for an 8-ary BVH, the compiler will continue to anticipate nodes sized at 120 bytes rather than 232 bytes, which will cause data misinterpretation.

The second approach to shader modularity is the use of preprocessor macros, which can be used for configurable arity of a BVH node, specialized control flow, and even interchangeable data types. However, with an increasing number of code specializations, the readability of the code decreases.

3.2 Slang

Slang shading language is a modern shading language designed to support compatibility with multiple back-ends while simultaneously providing functionalities to target a specific back-end. It provides capabilities for developing object-oriented code and utilizes a modular compilation approach to improve maintainability. Originally presented in the dissertation by Yong He^[27], it is currently hosted by the Khronos Group^[28].

Slang builds upon HLSL, a shading language commonly used in development with the DirectX API, and extends it by incorporating several modern programming features, including member functions and properties, operator overloads, interfaces, and generics^[29]. While GLSL shares similarities with C, Slang can be likened to C++; however, it differs in some aspects. Using

an object-oriented approach, the shader can be decomposed into shader modules, which encapsulate data and code specific to a particular feature and enable elegant shader specialization^[27].

The high level of modularity is also achieved by module precompilation and link-time specialization. Module precompilation works similarly to object file compilation. A set of source codes constitutes a module that provides access control and is processed separately. These modules can provide or use link-time constants and types: one module declares a constant or an interface of link-time type (optionally setting the default value), while another module defines and exports a constant value or a data type that implements the interface. During the linking process of a shader program, these specializations are resolved, resulting in appropriate optimization. The benefits of this approach include reduced compilation time and high readability of code because the entire code is type-checked using the interfaces^[29].

Although Slang supports many modern features, most of these features do not introduce any overhead. This is achieved through inlining and other optimizations performed at link time, when all specializations are already known. Unfortunately, resource allocation and performance are not as predictable as needed (see Section 5).

In general, Slang is a rapidly evolving shading language that offers many promising features. However, some of these features have not yet reached a high level of maturity, as indicated by the number of reported issues^[30] encountered while developing the traversal shader.

4 Implementation

In this section, the development aspects of the thesis are discussed. The implementation is done within the Orchard framework, which is detailed in the upcoming section. We implemented a wide BVH contraction shader with two heuristics, BVH traversal shaders in two languages (GLSL and Slang), and utilized two memory layouts.

4.1 Orchard

Orchard is a path-tracing framework designed primarily for research purposes, created by Martin Káčerik^[31]. The software is developed in C++ and uses the Vulkan API for communication with the GPU. All manipulation of BVH trees, from construction to traversal, is performed on the GPU using compute shaders.

In Orchard, BVH construction involves several phases: First, the initial BVH is built using the PLOC++ algorithm^[14], with one primitive per leaf. The following *Collapsing* step, using the method from the original PLOC paper^[13], merges some leaves to increase the number of triangles per leaf while decreasing the cost of the SAH. The subsequent *Transformation* phase, which is not used in wide BVH construction, changes the bounding volumes in the tree to different types. The last phase, *Rearrangement*, stores the BVH tree in a different memory layout used for traversal and possibly converts a binary BVH to a wide BVH.

In this memory layout for traversal, the bounding volume of a node is relocated to its parent. This allows the calculation of the ray-box intersection before the child node is added to the traversal stack, thus avoiding the need to load the child node if the ray does not intersect its bounding volume. Therefore, a binary BVH node contains two bounding volumes and two indices to child nodes, taking a total of 56 bytes, as shown in Figure 9.

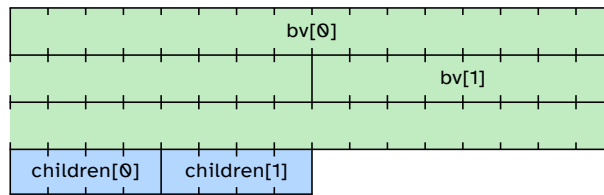


Figure 9: Memory layout of a binary BVH node, 56 bytes in total.

Path tracing utilizes the Wavefront approach^[32] and consists of three stages, as shown in Figure 10: the generation of primary rays, the BVH traversal that finds the nearest hit for all rays, and the shade + cast stage, where intersections are resolved and secondary rays are created based on the surface reflectance of a hit. Between each of these stages, GPU device synchronization is inserted. Currently, the framework supports only first-hit traversal using a diffuse (Lambertian) reflectance model, without any support for texture mapping.

BVH traversal shader is implemented via speculative while-while traversal with persistent threads (see Section 2.3). The traversal stack, consisting of 64 entries of 4 bytes each, is stored in

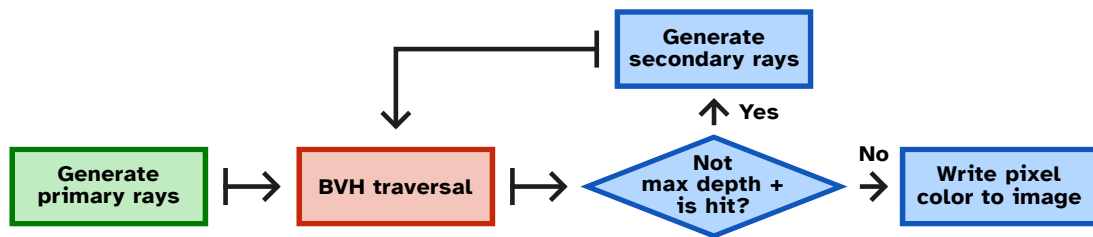


Figure 10: Diagram of wavefront shaders and their order. Each shader is represented by a different color. GPU barriers are shown as perpendicular lines at the start of each arrow.

local memory, whereas the stack top element and the leaf node awaiting ray-triangle intersection are stored in registers. The switch from BVH traversal to the ray-triangle intersection loop occurs only when every shader invocation in a subgroup has triangles to intersect. Additionally, new rays are fetched only if the number of active invocations is less than 20 (for clarification, one subgroup contains 32 invocations). This thesis did not delve deeply into configuring these parameters, as they appear to be well optimized and any tested alterations resulted in decreased MRps performance.

4.2 Wide BVH Construction

We implemented two variants of wide BVH construction, both based on the HPLOC contraction algorithm (see Section 2.1.4). They differ in the contraction heuristic used: one employs surface area, while the other uses view-dependent ray statistics.

4.2.1 Surface Area Contraction

For view-independent BVH contraction, a simplified greedy surface area heuristic is used (see Section 1.2.2). Given that the wide BVH construction proceeds from the top of the tree and all nodes considered for contraction share the same parent (or grandparent), we decided to simplify the formula by disregarding the division by the parent surface area:

$$C(N) = SA(N) \cdot P_N,$$

where P_N is the number of primitives in the node subtree. Due to the greedy nature of this heuristic, the upper tree structure is completely filled, whereas nodes closer to the leaves typically have fewer children, usually just two, because there are no nodes left for contraction.

4.2.2 View-dependent Contraction

For view-dependent contraction, the CBTC method of Gu et al.^[17] is employed (see Section 2.1.2). For the purpose of collecting ray traversal statistics, we enhanced the existing Orchard construction and tracing framework to support multiple Rearrangement and Trace passes, which can have different configurations. In this view-dependent contraction, the first pass is performed for the first sample of a view to gather ray-node statistics, while the second pass constructs the final BVH and traverses it for all the following samples.

The first pass constructs a basic binary BVH and allocates a buffer to store the visit count for each node, which is populated atomically by the binary tracer. Atomic variables are necessary because multiple shader invocations may access the same node concurrently. However, collecting these node statistics is significantly slower than standard BVH traversal. Gu et al.^[17] suggested that analyzing only a few thousand pixels is sufficient to achieve high-quality BVH contraction. To validate this, we implemented image subsampling (casting rays for every s -th pixel) and a depth limit on path tracing recursion in the ray statistics shader. As detailed in Section 6, our results confirm this claim.

After the first sample is traced, it switches to the second pass, and a final wide BVH is constructed via the CBTC method using the visit count buffer from the first pass. If the camera view is changed, the visit count buffer is reset and new view statistics are gathered. The BVH from the first pass is cached in memory and does not need reconstruction. However, since the final wide BVH depends on the view, it must be rebuilt.

4.3 Wide BVH Memory Layout

We implemented two wide BVH memory layouts: one utilizing uncompressed BVH nodes and another employing compression through quantization of bounding volumes.

The wide k -ary BVH nodes, in addition to storing child indices and bounding volumes analogous to those in binary BVH nodes, also contains a count of children, as their number can range from 2 to k . Initially, the binary BVH nodes included the primitive count of the entire subtree. Although this information was later removed from the binary BVH nodes, it remains present in the uncompressed wide BVH nodes.

4.3.1 Uncompressed Wide BVH Layout

The uncompressed wide BVH node includes all the variables previously described in the scalar layout, with variables aligned to the size of their scalar elements.

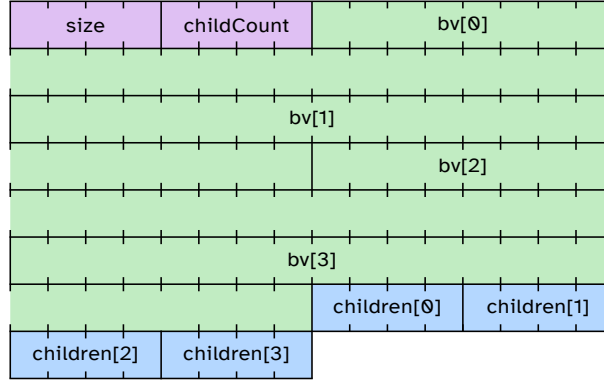


Figure 11: Memory layout of an uncompressed 4-ary BVH node, 120 bytes in total.

The AABB bounding volume is stored simply as two 3D vectors representing the corners. Child indices are 4-byte bit-field integers, as illustrated in Figure 12. The most significant bit (MSB) indicates whether the index represents an internal BVH node (MSB = 0) or a primitive range (MSB = 1). In the case of a primitive range, the lower 27 bits encode the index of the first primitive, while the subsequent 4 bits denote the number of triangles minus one. This encoding allows for up to 16 triangles within a single entity, under the assumption that a leaf node will never contain zero triangles.

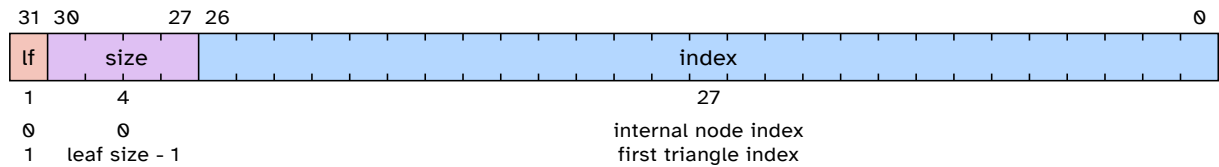


Figure 12: Encoding of a child index, representing either an internal node or a leaf which containing triangles.

A 4-ary BVH node requires 120 bytes, a 6-ary BVH node requires 176 bytes, and an 8-ary BVH node requires 232 bytes. The source code for this memory layout is shown in Listing 3, and Figure 11 provides a visual representation of this layout of a 4-ary BVH node.

```

1 struct NodeBvhWideTrace {
2     int32_t size;
3     int32_t childCount;
4
5     AABB bv[BVH_ARITY];
6     int32_t children[BVH_ARITY];
7 };

```

Listing 3: Definition of an uncompressed wide BVH node. BVH_ARITY represents the node's arity and is defined by a macro in GLSL or specialized via generics in Slang.

4.3.2 Quantized Wide BVH Layout

The quantized BVH node employs the same technique for computing the quantization grid and quantized bounding volumes as proposed by Ylitie et al.^[18] (see Section 2.2). The quantization grid, child internal node indices, and triangle indices are stored similarly, however, other variables are stored differently.

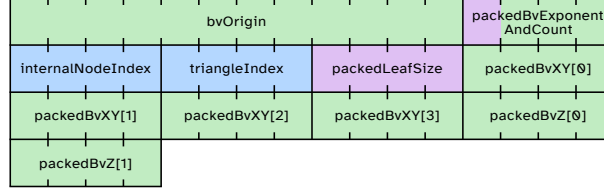


Figure 13: Memory layout of a quantized 4-ary BVH node, 52 bytes in total.

The children of a node are arranged so that all leaves precede the internal nodes. Consequently, it is unnecessary to store explicit information indicating whether a child is an internal node or a leaf. Instead, only the number of leaves l and the total number of children n are stored, each packed into one nibble (4 bits) within a single byte. This byte, together with the quantization grid exponents, is packed into a 4-byte integer, as shown in Figure 14. During traversal, it is known that the first l children are leaves, while all remaining children are internal nodes.

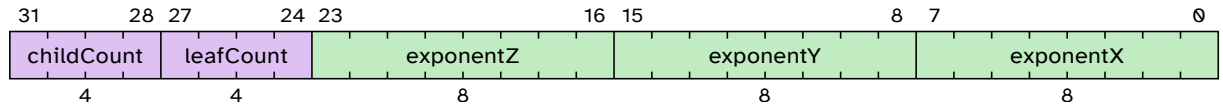


Figure 14: Encoding of quantization step exponents together with the counts of children and leaves into a 4-byte integer.

All internal child nodes and all triangles of the leaves are stored in own contiguous memory. The indices of the first internal child node and the first triangle are stored as 4-byte integers, while subsequent indices are computed by incrementing these initial indices as each child is processed. The count of triangles for all leaves is compressed into a 4-byte integer, with each leaf size stored as a nibble, starting from the least significant bit. The leaf size is encoded similarly to the uncompressed version (reduced by one).

Each quantized bounding volume requires 6 bytes. These are divided into a 4-byte integer representing the minimum and maximum values of the X and Y axes, and a 2-byte integer for the Z axis. Together with the Z axis from the second bounding volume, this forms a 4-byte integer that is stored in a separate array. Figure 15 illustrates this bounding volume packing. This approach differs significantly from the method of Ylitie et al.^[18], where each value of the bounding volumes for all children is packed inside an 8-byte variable, restricting it to 8-ary BVHs. In contrast, our method supports scaling the BVH arity (up to 8-ary) without altering the layout.

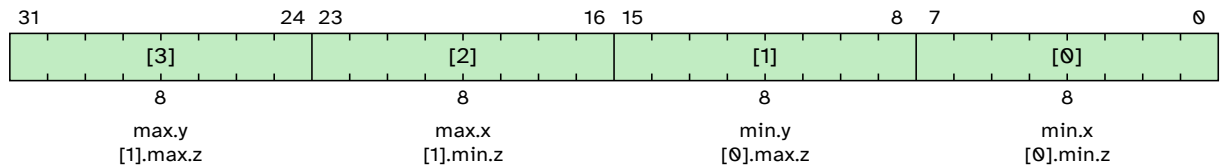


Figure 15: Encoding of a quantized bounding volume into one and a half 4-byte integers.

During traversal, each bounding volume is decompressed and the ray-box intersection is computed in world space, as opposed to the approach by Ylitie et al.^[18]. This decision was

primarily due to limited time, as transforming the ray into quantization space would have required additional code debugging.

A 4-ary BVH node requires 52 bytes (compression ratio $2.3\times$), a 6-ary BVH node requires 64 bytes (compression ratio $2.75\times$), and an 8-ary BVH node requires 76 bytes (compression ratio $3.05\times$). The source code defining this BVH node layout is shown in Listing 4, and Figure 13 displays the layout of a quantized 4-ary BVH node. This layout is 4 bytes smaller than the method of Ylitie et al.^[18], but our method does not support octant-based traversal order because we enforce the order of the children.

```

1 struct BvExponentAndCountPacked {
2     uint exponentX : 8;
3     uint exponentY : 8;
4     uint exponentZ : 8;
5     uint leafCount : 4;
6     uint childCount : 4;
7 };
8
9 struct NodeBvhWideTraceQuantized {
10     vec3 bvOrigin;
11     BvExponentAndCountPacked packedBvExponentAndCount;
12
13     uint32_t internalNodeIndex;
14     uint32_t triangleIndex;
15     uint32_t packedLeafSize;
16
17     uint32_t packedBvXY[BVH_ARITY];
18     uint32_t packedBvZ[BVH_ARITY / 2];
19 };

```

Listing 4: The definition of a quantized wide BVH node. BVH_ARITY represents the arity of a node and is defined by a macro in GLSL or specialized via generics in Slang. The BvExponentAndCountPacked structure is written using C bit-field syntax, which is supported in Slang.

4.4 Wide BVH Traversal

We developed an identical traversal shader using two shading languages, GLSL and Slang, in order to compare them. The traversal shader for wide BVHs follows the same principles as its binary BVH counterpart. The main differences lie in how a wide BVH node is processed and how the found intersections are stored within the traversal stack.

Children bounding volumes are intersected by the ray in a loop. Instead of loading an entire wide BVH node at once, variables are gradually loaded as required, since loading the entire wide BVH node would demand too many registers in the shader. To reduce latency associated with loading bounding volumes for intersection, the bounding volume is preloaded during the previous loop iteration. The distances from the ray origin to each intersection, together with the child indices, are stored in a temporary array and then sorted via a sorting network (see Section 2.3.4). The sorted intersected child indices are pushed to the traversal stack via a fall-through switch. To prevent a push-and-instant-pop sequence, the nearest intersected child is kept in registers.

With the BV-ray intersection loop fully unrolled (substituting the loop with its repeated body so that any control flow depending on the loop iteration variable is resolved statically), and with the sorting network and stack push switch accessing the temporary array solely through constant indices, the array can be fully stored in registers. Otherwise, storing this array in shared or local memory would be necessary, causing unnecessary memory traffic and resulting in significant performance penalties.

The GLSL and Slang versions of the traversal shader were written to be as similar as possible, without compromising readability or the use of Slang-specific features. The Slang codebase employs link-time specialization to enable a single traversal code that uses specialized functions for each wide BVH variant. All of these functions are inlined, and based on testing, these specializations do not introduce any performance slowdown.

However, GLSL and Slang exhibit notable behavioral differences, including variations in registers and shared memory allocations, instruction hoisting (moving part of the code to the top of a scope) performed by the Slang compiler, and a significant difference in MRps performance likely due to varying degrees of subgroup divergence. Based on these observed differences, a few patches for the Slang shader variant were proposed to maximize performance: introducing a workgroup control barrier between traversal and ray-triangle intersection while loops to enhance subgroup coherence (see Section 5.3), and modifying the ray-triangle intersection algorithm to precompute portions of the barycentric coordinates prior to evaluating any conditions (see Section 5.2).

Originally, the entire path tracing process (both binary and wide BVH) was implemented as a single kernel execution. All wavefront stages were executed within a single shader without employing a hardware scheduler. Instead, a software scheduler managed the stages by utilizing persistent threads and their ability to achieve device synchronization through atomic variables. After encountering difficulties described in detail in Section 5.1.1, we decided to switch to a separated kernel variant, where each wavefront stage is a standalone shader scheduled by hardware.

4.5 Support Tools

In addition to the wide BVH implementation, we developed several support tools to streamline the process: `nvidia-stabilize`, `compare-benchmark`, and `slang-depbuild`.

4.5.1 `nvidia-stabilize`

GPUs typically adjust core and memory frequencies dynamically to reduce power consumption during idle periods, which can cause inconsistencies in the measurement process. Manually setting fixed GPU clocks for every measurement session is tedious. Therefore, we developed the `nvidia-stabilize` tool to automatically stabilize GPU clocks during benchmark execution.

This tool serves as a wrapper for executing programs provided via command-line arguments: it configures the GPU, runs the specified program, and then restores the GPU to its initial state once the program completes. For GPU configuration, it uses the Nvidia Management Library (NVML)^[33]. Although this API is primarily designed for Nvidia Tesla GPUs, it also provides limited, but sufficient, support for consumer-grade graphics cards. Specifically, it sets persistence mode, locks the frequencies for the GPU’s graphics and memory components, and checks whether the GPU was throttled during benchmark execution. Moreover, by leveraging D-Bus^[34], it can deactivate the screen saver to mitigate any interruptions caused by the screen-saving procedure.

Determining the appropriate lock frequency is not as straightforward as it may seem. The base (non-boosted) frequency cannot be retrieved from the API, and the maximum graphics frequency reported for the tested GPU does not correspond to the highest achievable frequency in practice. After some experimentation, the following procedure was established: The program first attempts to set the maximum frequency reported by the API. It then waits a few seconds to ensure the GPU reflects the clock modification. Subsequently, it queries the current frequency f_{c1} and adjusts the final frequency slightly lower than this value, specifically to $0.98 \times f_{c1}$.

To ensure that CPU clocks remain constant throughout the benchmark, potential underclocking is detected by monitoring the violation status. The NVML API reports the total duration

during which the GPU was throttled. If this duration differs between the start and end of benchmark execution, the program reports that the measured results as potentially invalid.

4.5.2 compare-benchmark

We enhanced Orchard benchmarking capability to produce a comprehensive JSON file containing information about the GPU, all pipeline parameters, and measured build and trace times for each view and scene. This allows us to preserve the benchmark results for subsequent processing. Since this JSON contains a large amount of information, it is not feasible to read it manually, so we required a tool to process these JSON files.

compare-benchmark was originally designed to compare two benchmark files and output the differences in a human-readable format, specifically to produce commit messages, as shown in Figure 16, allowing us to track individual changes during development. It has since evolved beyond a comparison tool and can now convert a single benchmark JSON into multiple output formats: CSV tables with absolute or relative values for efficient chart creation, and summary LaTeX tables such as those in Section B.

```
stats: optimize basic traversal statistics collection
Performance changes from the last benchmark:
Wide4:
  Pass normal:
    pMRps: avg +2.29%, the best +2.65% in scene bistro_int, the worst +1.94% in scene lynxsdesign
    sMRps: avg +1.63%, the best +2.37% in scene san_miguel, the worst +1.12% in scene bistro_ext
Wide4 RDTc:
  Pass normal:
    pMRps: avg +2.23%, the best +2.43% in scene bistro_int, the worst +2.00% in scene lynxsdesign
    sMRps: avg +1.42%, the best +1.52% in scene red_autumn_forest, the worst +1.31% in scene bistro_int
  Pass node_stats:
    pMRps: avg +0.71%, the best +8.47% in scene bistro_int, the worst -5.78% in scene san_miguel
    sMRps: avg -0.57%, the best +3.50% in scene red_autumn_forest, the worst -5.72% in scene san_miguel
Wide4 Slang:
  Pass normal:
    pMRps: avg +2.85%, the best +4.07% in scene red_autumn_forest, the worst +2.03% in scene san_miguel
    sMRps: avg +7.47%, the best +12.49% in scene red_autumn_forest, the worst +1.67% in scene bistro_int
Wide6:
  Pass normal:
    pMRps: avg +2.12%, the best +2.75% in scene lynxsdesign, the worst +1.78% in scene bistro_int
    sMRps: avg +1.81%, the best +2.62% in scene bistro_int, the worst +1.00% in scene red_autumn_forest
Binary:
  Pass normal:
    pMRps: avg -0.52%, the best -0.28% in scene bistro_ext, the worst -1.08% in scene lynxsdesign
Wide8:
  Pass normal:
    pMRps: avg +0.72%, the best +1.17% in scene bistro_int, the worst +0.32% in scene red_autumn_forest
```

Figure 16: Example of a Git commit showing a shader comparison generated by the compare-benchmark tool.

This tool also enabled us to efficiently iterate on shader modifications and accurately quantify performance differences without creating new shader for each modification or manually comparing the values.

4.5.3 slang-depbuild

When we began developing Slang variants for wide BVH shaders, slangc, the official Slang compiler executable, did not support dependency file generation when using link-time specialization and module compilation. Dependency files indicate which source files are included, allowing the build system to trigger recompilation when any of these files change, thereby ensuring up-to-date compilation with minimal build actions.

Consequently, we developed an alternative compiler front-end, slang-depbuild, based on the Slang Compiler API, which produces a dependency file from the list of source files loaded by the compiler. It does not support all configuration options available in slangc, but only the subset we use. This enables Slang shaders to be compiled alongside other components via CMake.

5 Difficulties During Implementation

During the implementation of wide BVH shaders, we encountered several challenges. The following sections outline the most significant ones. Since some challenges are related to the GPU driver, it is important to note that all tests were conducted using the nvidia-open 570.124.04 driver.

5.1 Shader Resource Allocation

One of the main challenges we faced in implementing traversal shaders was the allocation of registers, shared memory, and local memory for each shader. GPUs impose several resource limits on the number of active threads, workgroups, and subgroups per compute unit, as well as on the number of registers and the size of shared memory. If any of these limits are exceeded, the total number of shader invocations that can run concurrently decreases. The ratio between the maximum possible and the actual number of active threads is called occupancy. Determining occupancy is a complex task due to the granularity at which resources are allocated and their partial interdependency [35].

These constraints may be simplified into more direct limits for the shader developer: the upper limits on registers (l_{reg}) and shared memory (l_{shared}) per shader invocation when occupancy is not yet constrained (in cases where the workgroup size is an integer fraction of the maximum number of active threads per compute unit). Table 1 presents these constraints for the evaluated GPU, the NVIDIA GeForce RTX 4070 Ti.

Registers per compute unit	65536
Shared memory per compute unit	49152 B
Subgroup size	32
Max workgroups per compute unit	24
Max workgroup size	1024
Max active invocations per compute unit	1536
Register allocation granularity	8
Subgroup allocation granularity	4
Shared memory allocation granularity	128 B
Max registers before occupancy decrease (l_{reg})	40
Max shared memory per invocation before occupancy decrease (l_{shared})	32 B

Table 1: Shader constraints and parameters of the evaluated GPU, NVIDIA GeForce RTX 4070 Ti. These data were obtained from the NVIDIA Nsight Compute Occupancy Calculator (NCOC) [36] and the Vulkan Physical Device Limit query [37]. While NCOC indicates that the GPU contains 102 kB of shared memory per compute unit, the shared memory limit in Vulkan is closer to 50 kB, as reported by the `maxComputeSharedMemorySize` value.

When the shader requires more than l_{reg} registers, the register allocator inside the GPU driver often decides to spill registers (temporarily storing them into memory), either to shared or local memory. This behavior may preserve the occupancy ratio, although it can increase memory traffic [38]. In certain instances, register spilling can enhance shader performance, while in other scenarios, such as some of ours, it may result in significant slowdowns. Although CUDA allows configuration of the register allocator [38], such options are absent in the Vulkan API.

Before detailing our issues, it is essential to highlight two separate expectations we had for the register allocator. The first is reasonable spilling of temporary registers (results of mathe-

matical calculations, loaded BVH node variables, etc.), and the second is allocating the traversal stack within local memory, because it is too large for registers and shared memory. As will be demonstrated in the following sections, we faced challenges with both expectations.

Method	Register count	Local memory size (bytes)	Shared memory size (bytes)
Binary (single)	48	256	3344
Binary	40	256	20
Binary (32×32)	48	256	140
Binary Slang	48	384	12
Wide4 (single)	48	256	3344
Wide4	48	256	12
Wide4 Slang (single)	156	0	4
Wide4 Slang	53	384	12
Wide6 (single)	64	256	4
Wide6	48	256	12
Wide6 (32×32)	53	256	12
Wide6 Slang	56	384	12
Wide8 (single)	64	256	4
Wide8	54	256	12
Wide8 Slang	62	384	12
Wide4/6/8 Quantized	48	256	1808
Wide4/6/8 Quantized (32×12)	56	256	12
Wide4 Slang Quantized	63	384	12
Wide6 Slang Quantized	68	384	12
Wide8 Slang Quantized	72	512	12

Table 2: Comparison of shader resource allocation. Shaders use the separated kernel version with a 32×2 workgroup size, unless otherwise specified. Expected memory usage in local memory (traversal stack) is 256 bytes (384 bytes for Slang), with 4 bytes in shared memory for the single (software scheduler) variant or 12 bytes in the separated kernel version (traversal statistics variables).

Selected benchmark shaders are in bold, and resource allocation issues are highlighted in red.

5.1.1 In Single Kernel Traversal

The traversal shader was initially implemented as a single kernel variant, integrating all Wavefront phases (see Section 4.1) within one persistent thread shader controlled by the software scheduler. This approach revealed problems with the register allocator, which can be categorized into two types based on the shading language: excessive register spilling in GLSL and excessive register allocation in Slang.

The path tracing task is highly demanding on memory traffic, mainly in uncompressed BVHs, generating many incoherent memory accesses. In such cases, a small drop in occupancy is not significantly problematic, as the primary bottleneck lies in memory speed. Consequently, unnecessary accesses caused by register spills negatively impact shader performance. Within

	Primary rays	Secondary rays
Binary	1.04	1.36
Wide4	1.16	1.67
Wide4 Slang	1.97	2.48
Wide6	1.00	1.00
Wide8	0.94	0.94

Table 3: The relative MRps (averaged across scenes) when comparing the separated kernel to the single kernel traversal shader with the same BVH arity (values greater than 1 indicate that the separated kernel is faster, each row is independent of the others).

the GLSL single kernel variant for binary and 4-ary BVHs, the register allocator performed extensive spilling, resulting in significant shader memory usage and limiting occupancy by shared memory size rather than register count (allocating 48 registers per 32×2 workgroup achieves 83% occupancy, however, using 3344 bytes of shared memory reduces occupancy to 58%).

In contrast, the single kernel variant implemented in Slang exhibits the opposite issue. The register allocator did not spill any registers, even the traversal stack was kept inside registers, leading to an extreme allocation of 156 registers and a very low occupancy of 25%. Table 2 presents the usage of registers, global, and shared memory for these and several other shaders.

During the analysis of these issues, we observed strange behavior in the binary GLSL single kernel traversal. Within the same program execution, the MRps performance of the binary traversal fluctuated every few seconds, varying by up to 100%. Our attempts to use the NVIDIA Nsight Graphics^[39] profiler to gain further insight into shader execution were largely unsuccessful. The first profile captured during path tracing produced different results compared to subsequent profiles. Figure 17 presents a screenshot comparing these two captured frame profiles. The exact cause of this behavior remains unclear. However, we hypothesize that the executed shader binary was somehow changed. Since both profiles originate from the same compiled Vulkan pipeline instance, it is likely that this modification occurs outside the application, possibly within the driver.

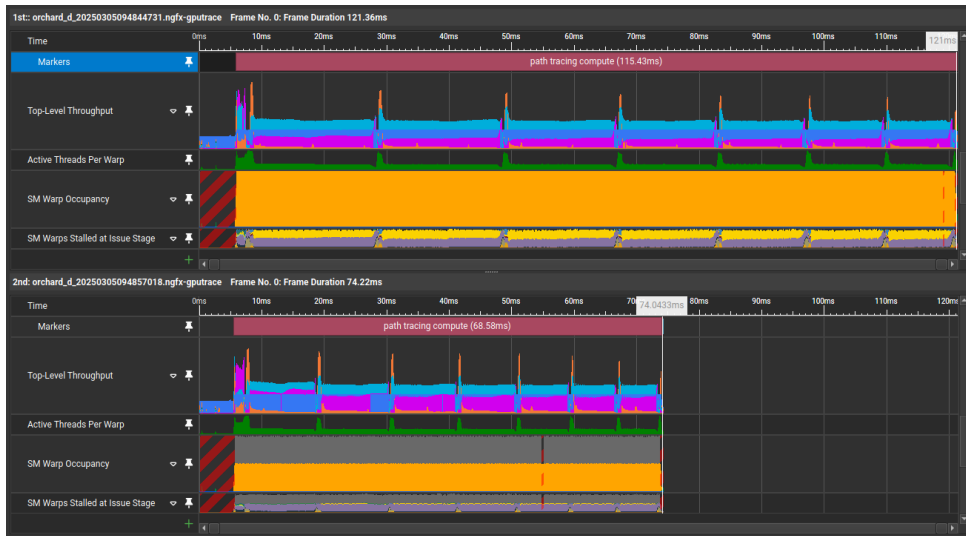
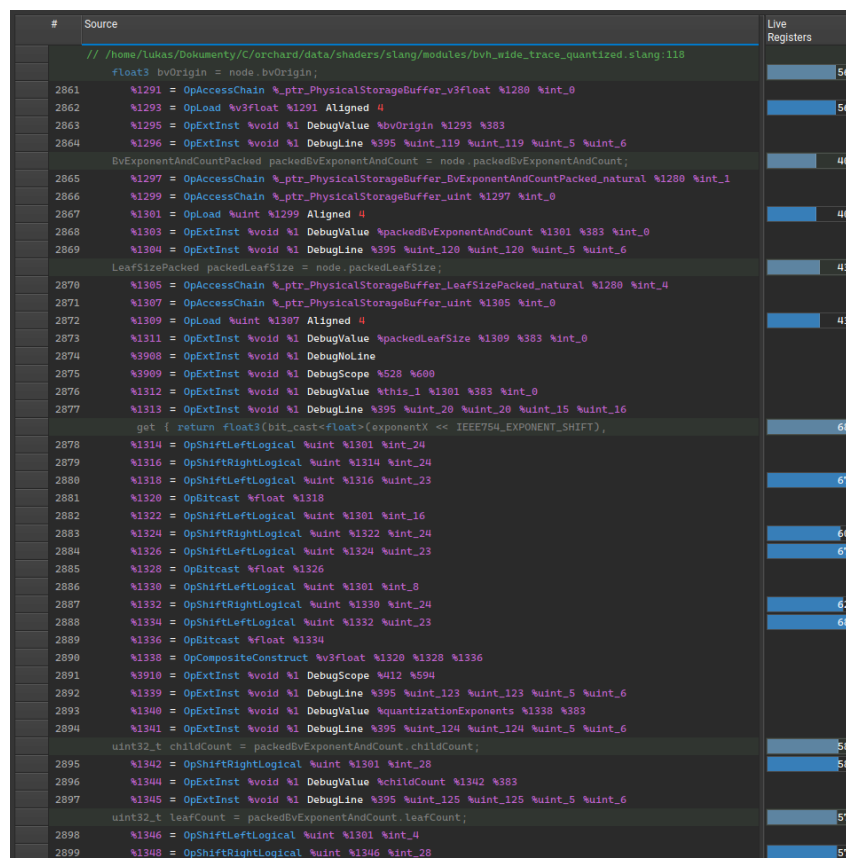


Figure 17: Comparison of two profile captures of the identical binary single kernel traversal shader during the same session using the NVIDIA Nsight Graphics^[39] profiler. The first profile indicates full occupancy, while the second shows occupancy limited to half, implying that the executed kernel was somehow altered, likely by the driver.

As a result of these problems, we decided to switch to a separate kernel architecture for the traversal shader, where each Wavefront phase is an independent shader. This change effectively resolved the issues of register spilling and performance fluctuations in the GLSL variant. The issue of excessive register allocation in the Slang variant was addressed by increasing the traversal stack size from 64 to 96 entries. Although this modification did not affect the single kernel variant, it ensured proper allocation of the traversal stack in local memory for the separated kernel variant. The Slang variant still requires slightly more registers than GLSL, but as demonstrated by the benchmark results in Section 6.7, this does not impact the MRPs performance of secondary rays.

5.1.2 In Quantized Wide BVH Traversal



The Slang implementation, similar to the uncompressed wide BVH, demands a large number of registers. Due to the reduced memory demands of the quantized wide BVH, shader occupancy becomes increasingly significant. The difference in the number of registers required between GLSL and Slang for the quantized 4-ary BVH (56 in GLSL versus 63 in Slang) is notable and reflected in the results (see Section 6.7).

This difference becomes even more significant in 6-ary and 8-ary BVHs, as the GLSL version demands a consistent number of registers regardless of BVH arity, whereas the Slang version requires up to 72 registers in the 8-ary BVH. This means 16 more registers, causing occupancy to decrease from 75% to 58%. The exact cause of this behavior remains unidentified, as NVIDIA Nsight Graphics^[39] shows atypical values of live registers, as depicted in Figure 18.

5.2 Instruction Hoisting in Slang

The original implementation of the Slang traversal shader exhibits contradictory results. In certain scenes (see Section 6.1 for a scene overview), the Slang variant showed a notable increase in MRps for secondary rays compared to the GLSL variant (+24% in `san_miguel`, +19% in `bistro_ext`, and +17% in `red_autumn_forest`), whereas in other scenes it was considerably slower (-19% in `lynxsdesign` and -30% in `bistro_int`). While investigating this problem, our analysis revealed an issue^[40] with instruction hoisting (moving part of the code to the top of a scope).

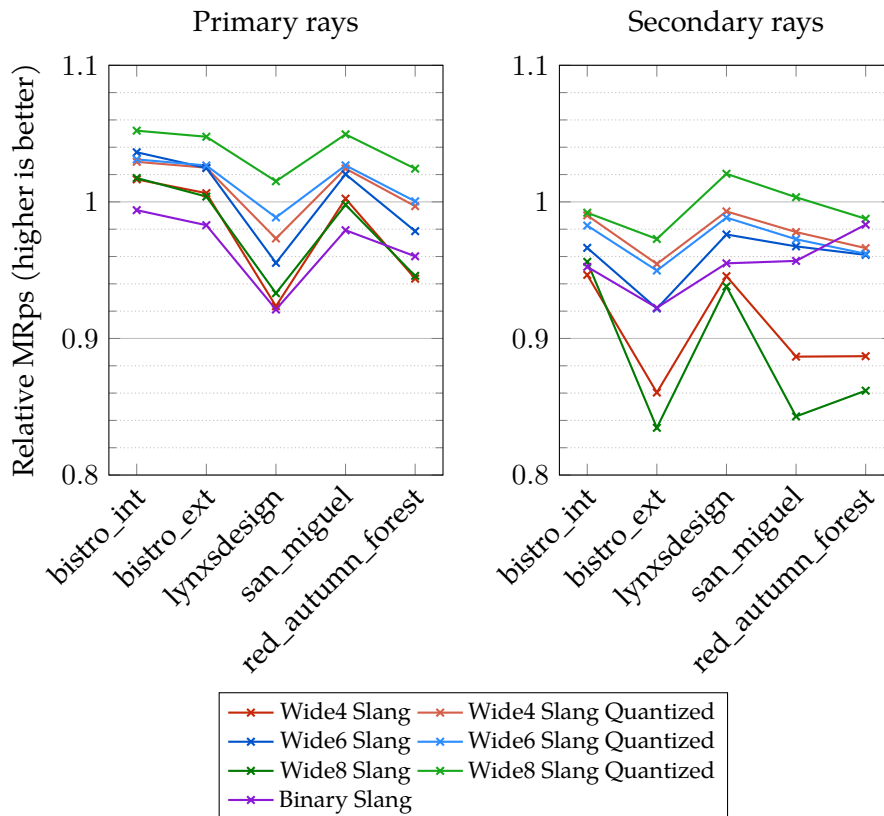


Figure 19: The relative MRps performance of Slang shaders compiled with version 2025.8 (including the instruction hoisting fix) compared to the same shaders compiled with version 2025.6.3 (before the fix).

In the ray-triangle intersection code, most of the barycentric coordinate calculations were moved prior to the condition checking the distance from the ray origin. We initially estimated that this mechanism decreases the duration between loading a variable from memory and its usage, and that addressing this issue would improve performance. However, fixing this issue (released in Slang 2023.6.4) surprisingly decreased the MRps performance of secondary rays by

approximately 5% on average (across all scenes and BVH variants), mainly in uncompressed 8-ary and 4-ary BVHs (-11.3% and -9.5%), as illustrated in Figure 19. Listing 5 shows the SPIR-V ray-triangle intersection code before and after this fix.

Insufficient time prevented us from comprehensively examine which other parts of the SPIR-V output were altered by this fix. However, we updated the ray-triangle intersection code to generate the same SPIR-V instructions as the Slang compiler produced prior to this fix, resulting in approximately a 3.8% MRps speedup of secondary rays on average, as illustrated in Figure 20. In contrast, implementing the identical ray-triangle intersection modification in the GLSL variant had a negligible effect.

1	%46 = OpCompositeExtract %float %37 3	1	%48 = OpCompositeExtract %float %37 3
2	%47 = OpVectorShuffle %v3float %37 %37 0 1 2	2	%49 = OpVectorShuffle %v3float %37 %37 0 1 2
3	%48 = OpDot %float %43 %47	3	%50 = OpDot %float %43 %49
4	%49 = OpFSub %float %46 %48	4	%51 = OpFSub %float %48 %50
5	%50 = OpDot %float %45 %47	5	%52 = OpDot %float %45 %49
6	%51 = OpCompositeExtract %float %39 3	6	%t = OpFDiv %float %51 %52
7	%52 = OpVectorShuffle %v3float %39 %39 0 1 2	7	%53 = OpFOrdLessThan %bool %t %float_0
8	%53 = OpDot %float %43 %52	8	OpSelectionMerge %54 None
9	%54 = OpFAdd %float %51 %53	9	OpBranchConditional %53 %55 %54
10	%55 = OpDot %float %45 %52	10	%54 = OpLabel
11	%56 = OpCompositeExtract %float %41 3	11	%56 = OpCompositeExtract %float %39 3
12	%57 = OpVectorShuffle %v3float %41 %41 0 1 2	12	%57 = OpVectorShuffle %v3float %39 %39 0 1 2
13	%58 = OpDot %float %43 %57	13	%58 = OpDot %float %43 %57
14	%59 = OpFAdd %float %56 %58	14	%59 = OpFAdd %float %56 %58
15	%60 = OpDot %float %45 %57	15	%60 = OpDot %float %45 %57
16	OpSelectionMerge %61 None	16	%61 = OpFMul %float %t %60
17	OpSwitch %int_0 %62	17	%u = OpFAdd %float %59 %61
18	%62 = OpLabel	18	%62 = OpFOrdLessThan %bool %u %float_0
19	%t = OpFDiv %float %49 %50	19	OpSelectionMerge %63 None
20	%63 = OpFOrdLessThan %bool %t %float_0	20	OpBranchConditional %62 %64 %63
21	OpSelectionMerge %64 None	21	%63 = OpLabel
22	OpBranchConditional %63 %65 %64	22	%65 = OpCompositeExtract %float %41 3
23	%64 = OpLabel	23	%66 = OpVectorShuffle %v3float %41 %41 0 1 2
24	%66 = OpFMul %float %t %55	24	%67 = OpDot %float %43 %66
25	%u = OpFAdd %float %54 %66	25	%68 = OpFAdd %float %65 %67
26	%67 = OpFOrdLessThan %bool %u %float_0	26	%69 = OpDot %float %45 %66
27	OpSelectionMerge %68 None	27	%70 = OpFMul %float %t %69
28	OpBranchConditional %67 %69 %68	28	%v = OpFAdd %float %68 %70
29	%68 = OpLabel	29	%71 = OpFOrdLessThan %bool %v %float_0
30	%70 = OpFMul %float %t %60	30	OpSelectionMerge %72 None
31	%v = OpFAdd %float %59 %70	31	OpBranchConditional %71 %73 %74
32	%71 = OpFOrdLessThan %bool %v %float_0	32	%74 = OpLabel
33	OpSelectionMerge %72 None	33	%75 = OpFAdd %float %u %v
34	OpBranchConditional %71 %73 %74	34	%76 = OpFOrdGreaterThan %bool %75 %float_1
35	%74 = OpLabel	35	OpBranch %72
36	%75 = OpFAdd %float %u %v	36	%73 = OpLabel
37	%76 = OpFOrdGreaterThan %bool %75 %float_1	37	OpBranch %72
38	OpBranch %72	38	%72 = OpLabel
39	%73 = OpLabel	39	%77 = OpPhi %bool %76 %74 %true %73
40	OpBranch %72	40	OpSelectionMerge %78 None
41	%72 = OpLabel	41	OpBranchConditional %77 %79 %78
42	%77 = OpPhi %bool %76 %74 %true %73		
43	OpSelectionMerge %78 None		
44	OpBranchConditional %77 %79 %78		

(a) SPIR-V assembly before the fix

(b) SPIR-V assembly after the fix

Listing 5: Comparison of SPIR-V assembly for the ray-triangle intersection code before and after applying the instruction hoisting fix. Conditional branch instructions are highlighted in red. As shown, the variables *u* and *v* are no longer partially precomputed (lines 6 to 15 in 5a) before the condition (line 22 in 5a).

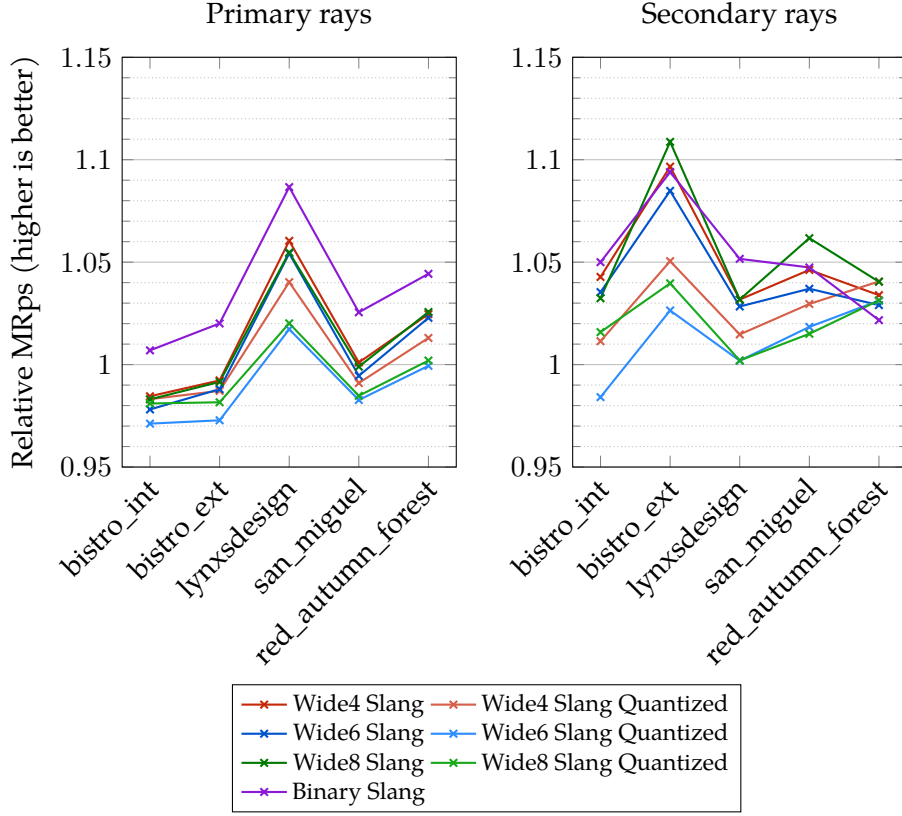


Figure 20: The relative MRps performance of Slang shaders with the ray-triangle intersection modification compared to the same shaders without this modification.

5.3 Subgroup Coherency in Slang

Another difference we identified between the GLSL and Slang variants was the differing average number of active threads per subgroup throughout shader execution. In the Lynxdesign interior scene, where the Slang variant exhibited the lowest relative performance against GLSL, the average Slang coherence in the 4-ary BVH was 10.3 (out of 32 threads per subgroup), while GLSL achieved 12.9. This smaller subgroup coherence resulted in more memory access stalls for the Slang variant, thus reducing its performance.

Because both variants use the same while-while traversal loop with identical parameters for exiting these loops, we hypothesize that this difference arises from invocations that do not properly reconverge. Initially, we tried enabling the `SPV_KHR_maximal_reconvergence` extension^[41], but it had no effect on MRps performance. Therefore, we decided to use explicit synchronization between the BVH traversal and ray-triangle intersection loops. The most granular synchronization available is the control and memory barrier of a workgroup, implemented by `GroupMemoryBarrierWithGroupSync`, as our attempts to synchronize only a subgroup using the inlined `OpControlBarrier` SPIR-V instruction caused a crash in the Slang compiler.

This barrier caused a rise in subgroup coherence, for example, in the Lynxdesign interior scene, reaching an average of 13.8 in the 4-ary BVH. Performance comparison was performed with fine-tuned workgroup sizes for each shader variant of both (with and without barrier) versions, as presented in the tables in Section A. For secondary rays, it caused an increase of approximately 9–11% on average for uncompressed BVHs and 24–27% for quantized BVHs. However, it did not improve performance in all cases, especially in scenes where the Slang variant showed higher secondary ray performance than GLSL (specifically `bistro_ext`, `san_miguel`, and `red_autumn_forest`), with a worst case of -15% in a binary BVH in the Lumberyard Bistro exterior scene. For primary rays, changes were more stable, ranging from -0.6% (8-ary uncompressed)

to +10% (binary uncompressed). Figure 21 illustrates the changes in MRps performance for quantized and uncompressed BVH variants across each test scene.

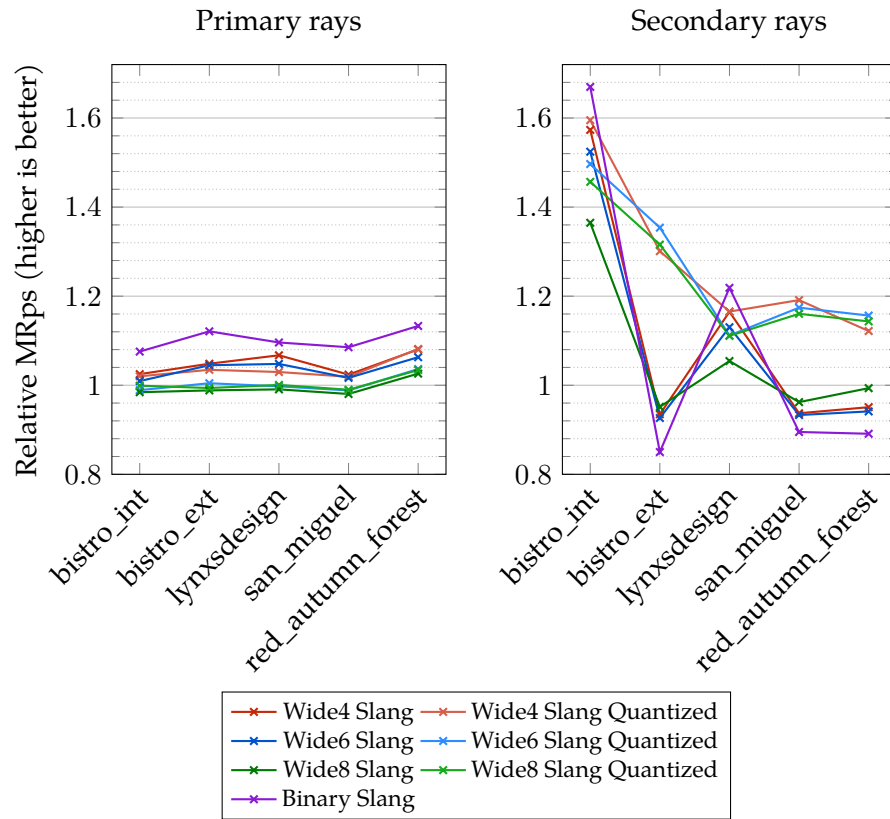


Figure 21: The relative MRps performance of Slang shaders with a workgroup barrier between BVH traversal and triangle intersection loops, compared to the same shaders without this barrier.

6 Results

This section presents a detailed evaluation of the combinations of implemented BVH construction methods (SAH and CBTC; see Section 4.2), memory layouts (uncompressed and quantized; see Section 4.3), traversal shaders (GLSL and Slang; see Section 4.4), and arity (4-ary, 6-ary, 8-ary). Together with the reference binary GLSL shader and the newly developed Slang variant of binary BVH traversal, there are 26 combinations in total. Given this large number of combinations, we perform comparisons in segments rather than all at once. Furthermore, due to the extensive number of values for each measurement, this section mostly presents aggregate values or optimal parameter configurations, with more detailed tables and charts available in the appendix.

The names of the shader variants follow this pattern: `<Arity> [Slang] [Quantized] [CBTC]`. Therefore, unless specifically referred to as Slang/Quantized/CBTC, it refers to the GLSL/Uncompressed/SAH variant. A binary BVH variant is termed Binary, while 4-ary, 6-ary, and 8-ary wide BVHs are labeled Wide4, Wide6, and Wide8, respectively.

The performance of path tracing, measured in mega rays per second (MRps), is influenced by the configuration of various parameters. Specifically, we focus on the configuration of the work-group size (separately for primary and secondary rays), the SAH traversal cost c_t for the leaf compaction step and depth limit, the number of rays, and the RDTC/SATC threshold for view-dependent BVH construction.

6.1 Dataset

Our benchmarking dataset consists of five scenes. Three of these are well-known example scenes: the interior and exterior of the Lumberyard Bistro^[42], and San Miguel 2.0^[43], Morgan McGuire’s modification of the original version by Guillermo M. Leal Llaguno. The other two scenes are Blender splash screens: Red Autumn Forest^[44] by Robin Tran, and an interior room by Lynxsdesign^[45]. For each scene, we selected eight different camera views. Figure 22 presents the selected rendered view for each of these scenes.



Figure 22: Example view for each testing scene: (a) San Miguel 2.0^[43], (b) Lumberyard Bistro interior^[42], (c) Lumberyard Bistro exterior^[42], (d) Lynxsdesign’s interior room^[45], and (e) Red Autumn Forest^[44]. Images were rendered using 128 samples per pixel, except for (c), which uses 512 samples per pixel, and edited with Adaptive Histogram Equalization.

6.2 Testing Environment

The benchmark was performed on a Linux desktop system with the following specifications and software installed:

1. **CPU:** Intel Core i5-9600K @ 4.5 GHz
2. **GPU:** Nvidia GeForce RTX 4070 Ti @ locked 2835 MHz GPU / 10501 MHz MEM
3. **Linux kernel:** 6.13.6
4. **Nvidia driver:** nvidia-open 570.124.0
5. **Slang version:** 2025.8
6. **glslang version:** 15.1.0

To ensure stable GPU performance, we fixed the GPU core clock at 2835 MHz and the GPU memory clock at 10501 MHz using the `nvidia-stabilize` utility (see Section 4.5.1). Each scene was rendered from 8 different views, with 15 path samples collected per pixel per view. The resulting values are the average taken over all path samples and views within the scene.

6.2.1 Workgroup Size

The workgroup size is one of the most important parameters for GPU BVH traversal. Since shader invocations in the GLSL variant do not communicate with other threads, the impact of workgroup configuration should be minor when the GPU is adequately saturated. However, in practice, the difference in performance for the same shader executed with various workgroup sizes can be as high as 35%. Tables 7 and 8 show the relative MRps performance of GLSL shaders for primary and secondary rays. Due to significant scene variations under certain conditions, the optimal workgroup size was determined by the largest minimum value across all scenes, considering primary and secondary rays independently. Table 4 presents the workgroup sizes chosen for each traversal shader variant (GLSL and Slang).

		Uncompressed		Quantized	
		GLSL	Slang	GLSL	Slang
Binary	Primary	32×20	32×2	X	
	Secondary	32×32	32×2		
Wide4	Primary	32×20	32×2	32×12	32×2
	Secondary	32×2	32×2	32×12	32×2
Wide6	Primary	32×2	32×2	32×20	32×2
	Secondary	32×32	32×2	32×12	32×2
Wide8	Primary	32×2	32×2	32×20	32×2
	Secondary	32×2	32×2	32×12	32×2

Table 4: Overview of the selected workgroup sizes across all shader variants.

The primary reason why the workgroup size leads to significant variations is mainly due to the different allocation of resources (registers, shared and local memory), as described in Section 5.1. It cannot be said that lower register and shared memory usage always leads to the best possible performance. For example, the binary GLSL variant requires 40 registers and 20 bytes of shared memory in the configuration with the 32×2 workgroup, but exhibits approximately 10% lower MRps performance for secondary rays compared to the 32×32 workgroup, which requires 48 registers and 140 bytes of shared memory. We hypothesize that this can probably be caused by multiple reasons. The varying sizes of workgroups may result in different GPU occupancy because certain workgroup sizes do not align with an integer fraction of the maximal

number of active subgroups. Because of this, the register allocator in the driver can distribute more registers before reaching the threshold, which reduces the number of workgroups that can fit into a compute unit. The second reason is that reduced GPU occupancy can help the memory subsystem.

In Slang shaders, we use workgroup synchronization, as described in Section 5.3. Consequently, the best workgroup size differs significantly from the GLSL version. With an increase in workgroup size, the number of subgroups needing synchronization also rises, leading to increased idle time as some subgroups wait for others to complete the BVH traversal loop. Therefore, for all Slang variants, the best workgroup size is 32×2 , as illustrated in Tables 9 and 10. To compare the advantages of the workgroup barrier in Slang, the optimal workgroup size was also identified for the version without the barrier. Tables 11 and 12 demonstrate that for all shader variants, except primary rays of uncompressed 4-ary (32×20) and 6-ary (32×12) BVH, the optimal workgroup size is 32×32 . It is worth mentioning that the values in each of the tables are relative to the optimal workgroup size of a particular shader variant and do not present any relation between multiple shader variants.

6.3 SAH Traversal Cost

The SAH cost is used to merge leaf nodes in *Collapsing step* (see Section 4.1), resulting in leaves containing a maximum of 16 triangles each. To find the best SAH cost parameters for each BVH variant, we fixed the intersection cost $c_i = 2$ and varied only the traversal cost c_t to reduce the size of the explored state space.

$c_i = 2$	Uncompressed		Quantized	
	GLSL	Slang	GLSL	Slang
Binary	$c_t = 2$	$c_t = 2$	X	
Wide4	$c_t = 2$	$c_t = 2$		
Wide6	$c_t = 3$	$c_t = 3$	$c_t = 3$	$c_t = 4$
Wide8	$c_t = 4$	$c_t = 4$	$c_t = 3$	$c_t = 4$

Table 5: Overview of the selected SAH traversal cost c_t across all shader variants.

With increasing arity of the BVH, the optimal c_t increases accordingly. For uncompressed binary and 4-ary BVHs, the optimal value for c_t is 2, while for 6-ary it is 3, and for 8-ary it is 4. The quantized BVHs behave slightly differently. The GLSL quantized variant inclines toward a smaller c_t value, specifically in the 8-ary BVH, where the optimal $c_t = 3$. This is probably due

$c_i = 2, c_t =$	GLSL						Slang					
	1	2	3	4	5	6	1	2	3	4	5	6
Binary	0.99	1.00	1.00	1.00	0.99	0.98	0.99	1.00	0.99	0.99	0.97	0.96
Wide4	0.99	1.00	1.00	0.99	0.98	0.96	0.99	1.00	0.99	0.99	0.97	0.95
Wide4 Quantized	0.99	1.00	0.99	0.97	0.95	0.92	0.98	1.00	1.00	0.99	0.97	0.95
Wide6	0.98	1.00	1.00	1.00	0.99	0.98	0.99	1.01	1.00	0.99	0.98	0.97
Wide6 Quantized	0.99	1.00	1.00	0.99	0.97	0.95	0.96	0.99	1.00	1.00	1.00	0.99
Wide8	0.97	1.00	1.00	1.00	0.99	0.98	0.96	0.99	0.99	1.00	1.00	0.99
Wide8 Quantized	0.97	1.00	1.00	0.99	0.98	0.97	0.95	0.98	0.99	1.00	1.00	1.00

Table 6: Average relative MRps of the scenes using varying c_t parameters for leaf node compaction. The best MRps for each shader variant is highlighted in bold.

to the smaller memory footprint of the BVH nodes, which makes it faster to process more BVH nodes than larger leaves. In contrast, the Slang variant of quantized BVH requires larger leaves ($c_t = 4$ for 6-ary BVH), likely due to increased register demands. Table 5 presents the optimal c_t for each shader variant, while Table 6 shows the relative average MRps performance across all tested c_t values.

6.4 CBTC Heuristic Threshold

The first parameter examined in the construction of view-dependent BVH is the node visit count threshold t , beyond which the CBTC heuristic is replaced by the SATC heuristic. We decided to represent this value as an absolute number rather than a ratio of sample rays, since we want to set a lower bound for statistical precision (because the visit count is an integer, the ratio of visit counts can only take on a limited set of values when the count is low).

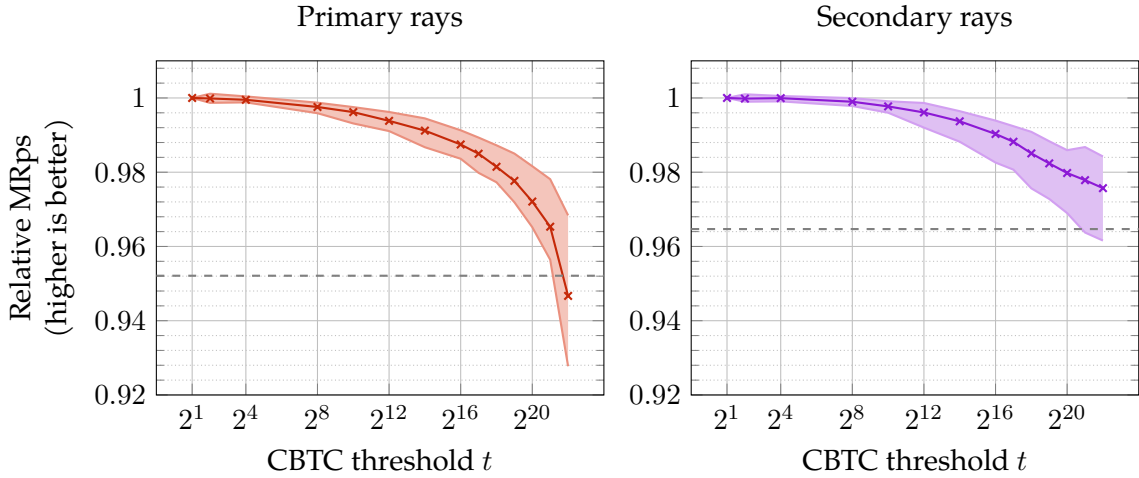


Figure 23: Relation between the CBTC heuristic t threshold and the relative MRps performance of the resulting BVH. The solid line illustrates the average across all scenes and shader variants, while the colored surface represents the range from the minimum to maximum values of the shader variants’ scene averages. The dashed line represents the average MRps of shader variants for non-view-dependent BVHs.

For this test, we collected samples from a single fully rendered frame (1920×1080 primary rays and their bounced secondary rays up to depth 8). As shown in Figure 23, MRps performance remains nearly constant for small threshold values t (up to around 16). However, when t exceeds this range, MRps performance begins to decrease. This is due to a reduction in the number of nodes contracted through the CBTC heuristic, causing a fallback to the SATC method. Consequently, we selected the threshold value of $t = 8$ to avoid an unnecessarily low value while ensuring optimal MRps performance.

6.5 Depth Limit for Ray Statistics

Since all meshes exhibit solely diffuse reflection and secondary rays tend to be distributed rather randomly, we aimed to evaluate whether limiting the maximum depth of secondary rays d influences the quality of the constructed view-dependent BVH. $d = 0$ indicates only primary rays without secondary rays, while $d = 7$ implies unrestricted conditions equivalent to our standard path tracing. Similarly to the CBTC heuristic threshold analysis, sample rays were obtained from a 1920×1080 frame, and contraction was performed using the CBTC threshold $t = 8$.

Figure 24 presents the relative MRps for both primary and secondary rays, along with the relative duration of the ray sampling process. For primary rays, constraining the depth d does not degrade MRps performance. In fact, depth limits of 2 and 1 yield a marginal improvement

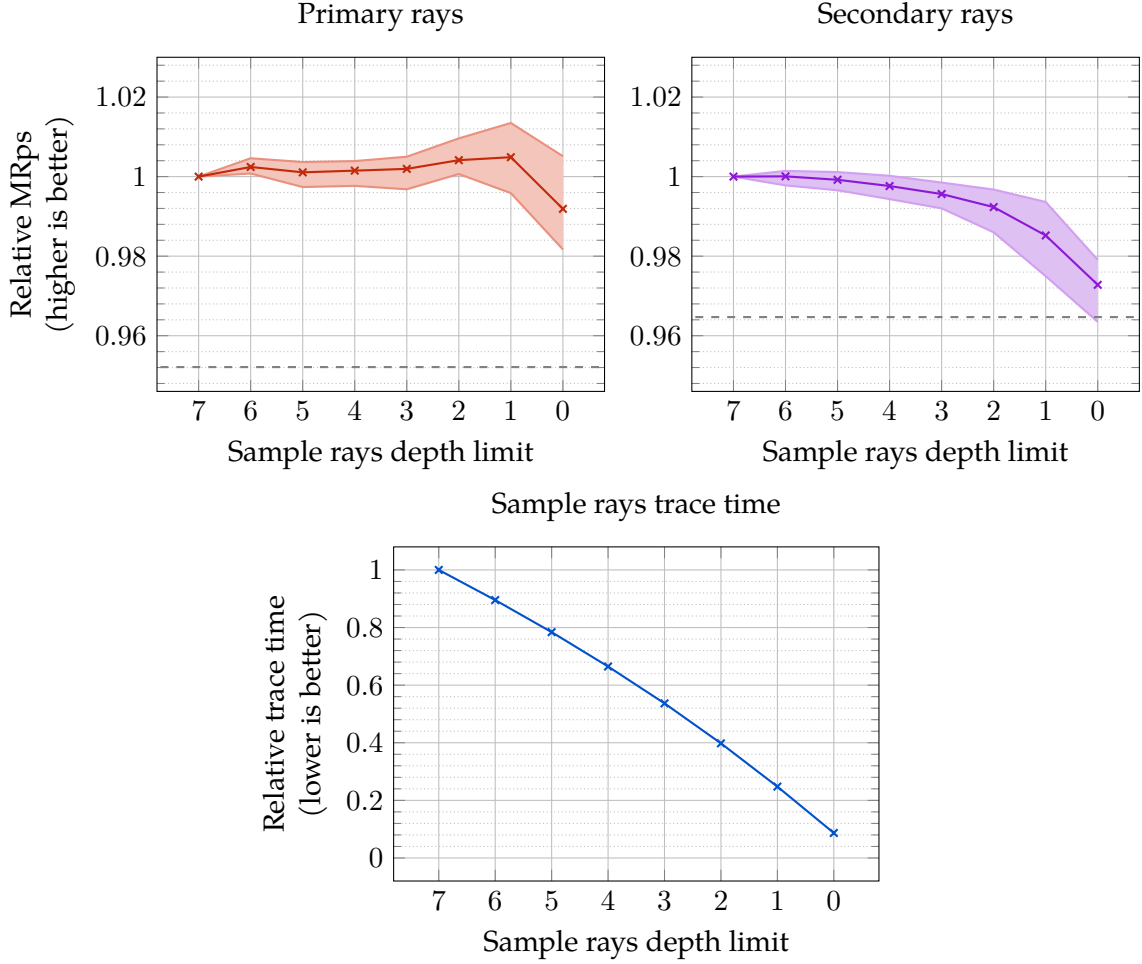


Figure 24: Relation between the depth limit d of sample rays and relative MRps performance on the resulting BVH. The solid line illustrates the average across all scenes and shader variants, while the colored surface represents the range from the minimum to maximum values of the shader variants’ scene averages. The dashed line represents the average MRps of shader variants for non-view-dependent BVHs.

of around 0.5%. However, omitting secondary rays entirely ($d = 0$) leads to an approximate 1% decrease. For secondary rays, limiting the ray depth gradually decreases performance, up to -3% at $d = 0$. The trace time of the sample rays decrease roughly linearly with the depth limit. Therefore, we chose the depth limit for the final benchmark as $d = 4$, since the MRps of secondary rays is only marginally smaller (by 0.25%), but the time to collect sample rays decreases to 66%.

6.6 Number of Sample Rays

Gu et al.^[17] stated that for a high-quality contracted BVH, it is sufficient to sample only around 0.1%–0.5%. We aim to confirm this statement. For the sample rays frame, we employ subsampling: rather than casting rays for every pixel, we cast rays for only every s -th pixel (therefore, the number of cast rays is reduced by a factor of $\frac{1}{s}$). This test was performed at a resolution of 1920×1080 , with $t = 8$ and $d = 7$.

As shown in Figure 25, the MRps performance decreases only marginally (about -0.5% for $s = 10 \times 10$, and -0.6% for $s = 256$), while the time required to trace sample rays decreases rapidly (to only 2.5% of the original time for $s = 10 \times 10$). For very small numbers of cast rays (particularly when $s \geq 64$), the trace time slowly stops decreasing because the GPU is no longer

fully saturated. Consequently, we chose $s = 64$ (about 1.6% of rays cast), although the range of 0.1%–0.5% proposed by Gu et al. is also applicable.

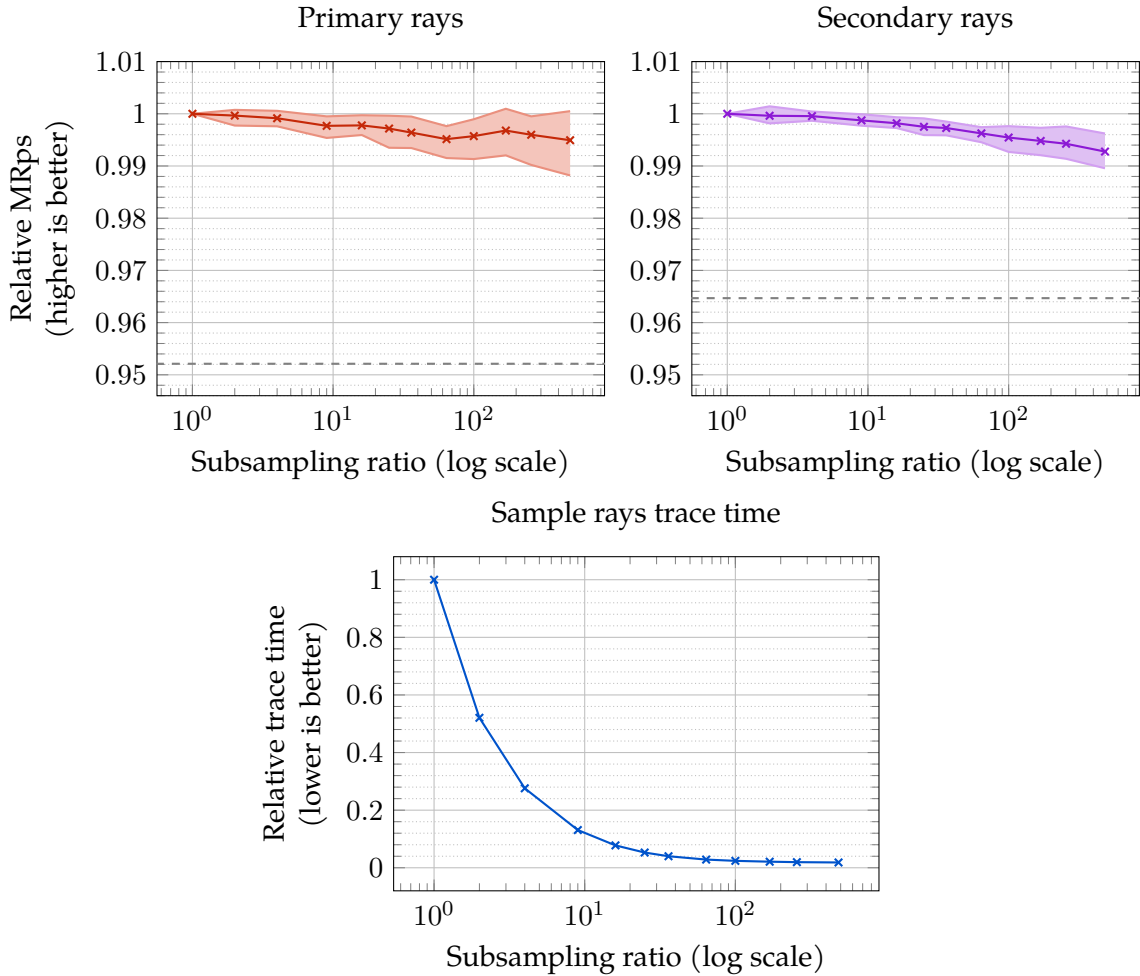


Figure 25: Relation between subsampling ratio s (casting $\frac{1}{s}$ times fewer rays) and relative MRPs performance on the resulting BVH. The solid line illustrates the average across all scenes and shader variants, while the colored surface represents the range from the minimum to maximum values of the shader variants’ scene averages. The dashed line represents the average MRPs of shader variants for non-view-dependent BVHs.

6.7 Overall Results

Figure 26 presents the average results across all shader variants and scenes, grouped by BVH arity. Detailed tables for each scene are provided in Section B, and Figures 27–30 show the relationships between uncompressed, quantized, and view-dependent variants on a per-scene basis. For the majority of shader variants, the 4-ary BVH consistently achieves the best average scene performance, with the only exception being the 6-ary quantized BVH. For 4-ary BVHs, the number of visited nodes decreases to approximately 50%, along with bounding volumes tested (up to -6%) and triangles tested (up to -3%).

For BVHs with higher branching factors, the number of visited nodes decreases more slowly because nodes near the leaves are often not fully occupied, containing only 2 or 3 children. Consequently for 6-ary and 8-ary BVH, the number of bounding volumes (up to +23%) and triangles (up to +22%) tested increases, as BVH contraction removes some nodes that previously pruned the traversal tree.

Uncompressed view-independent BVHs exhibit a +20% speedup for secondary rays in 4-ary BVH, +15% in 6-ary BVH, but only +2% in 8-ary BVH. For primary rays, only the 4-ary BVH is faster (+7%) than the binary BVH, while the 6-ary and 8-ary variants are equal or slower (-11%). The overhead of BVH contraction is nearly negligible: +2% for 4-ary BVH and even -1% for 8-ary. The Slang variant provides slightly better MRps performance for 4-ary and 6-ary BVHs (+1% and +2.5%, respectively), but shows -7% lower MRps for the 8-ary BVH. For primary rays, Slang shaders perform much worse (-17% to -20%).

One testing scene deviates notably from the others: Lynxsdesign’s interior. It contains highly subdivided meshes (e.g., sofas), while the rest of the room consists of flat surfaces with very few triangles. This is reflected in the counts of triangles and bounding volumes tested. Many shader variants indicate only a minor performance increase or even a decrease relative to the binary BVH, particularly for Slang shaders (up to -39% in Wide8 Slang Quantized). This behavior is not viewed as a poor choice of scene, but rather as an example where the proposed method does not perform optimally in every situation.

Quantization of wide BVHs appears highly beneficial. Secondary rays are traced approximately 48% faster for 4-ary, 55% faster for 6-ary, and 58% faster for 8-ary BVHs compared to their uncompressed counterparts (+74%, +78% and +61% compared to binary BVH). This speedup varies by scene: for lynxsdesign it averages only 5%, and for bistro_int it reaches only 20%. However, for primary rays, which are more coherent and less sensitive to memory latency, MRps performance decreases by about 16–18%. Due to BVH compression, construction time increases approximately 8–12%. Quantization can also slightly enlarge bounding volumes, resulting in 2.5–4% more volumes tested.

The Slang variant of quantized BVHs is significantly slower, as discussed in Section 5.1.2: approximately -9% for 4-ary, -14% for 6-ary, -12% for 8-ary BVH in secondary rays, and -18% to -28% for primary rays. The difference in the number of tested triangles and bounding volumes between the 6-ary and 8-ary Slang versus GLSL variants is due to the different optimal traversal cost parameter c_t , as shown in Table 5.

Performance improvement in view-dependent BVHs was less than expected. The difference between view-dependent and independent BVHs is quite similar for both uncompressed and quantized variants. For the 4-ary BVH, the improvement is approximately 4% for secondary rays and 9% for primary rays, mainly due to an outlier in the form of the san_miguel scene, where primary rays show a 30% increase in MRps.

As the branching factor increases, the improvement decreases. For the 6-ary BVH, the improvement is only 5% and 1% for primary and secondary rays, respectively. For the 8-ary BVH, secondary rays are even 2% slower than in the view-independent BVH. In terms of bounding volumes tested, the 4-ary BVH tests around 5% fewer volumes, and the 6-ary around 4% fewer. The 8-ary BVH tests roughly the same number of bounding volumes, corresponding to the lack of performance increase. The construction overhead from tracing sample rays and rearranging two BVHs (a binary BVH for sample rays and the final wide BVH) instead of just one is about 19–21%.

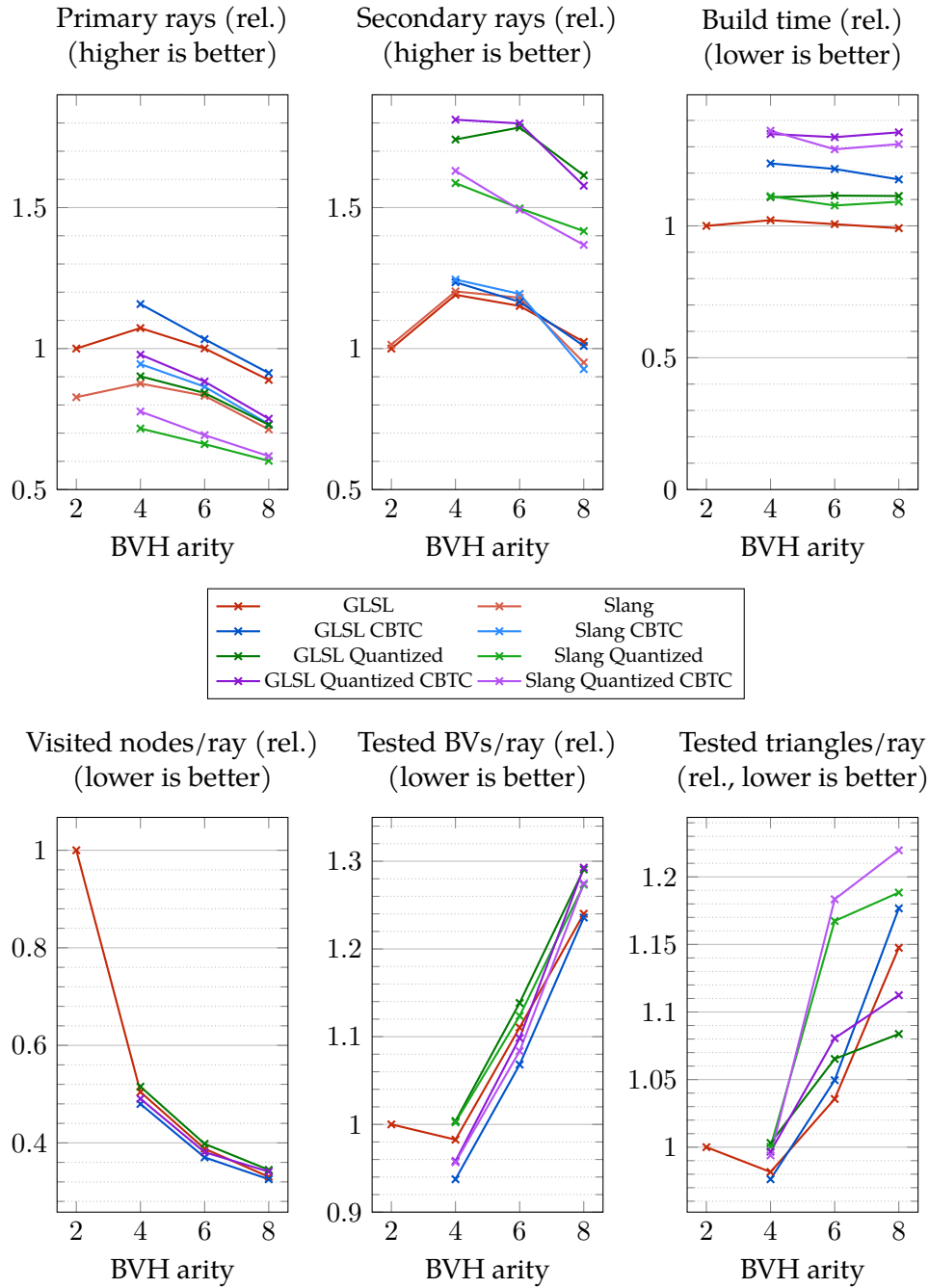


Figure 26: Benchmark results for all scenes. All values are relative to the Binary variant and averaged across the testing scenes. Build time includes the total time taken to construct all BVHs and the time spent casting sample rays. In certain charts, the Slang shader variants are not included because their values are very similar to those of the GLSL variant.

7 Conclusion and Future Work

We implemented wide BVH construction using two heuristics (SAH and view-dependent), two memory layouts (uncompressed and quantized bounding volumes), and evaluated their traversal using two shading languages (GLSL and Slang). Across five different testing scenes, the best results in each category were shown by: 4-ary uncompressed BVH (+20%), 4-ary view-dependent uncompressed BVH (+24%), 6-ary quantized BVH (+78%), and 4-ary view-dependent quantized BVH (+81%).

The view-dependent BVH construction shows less performance increase than anticipated (only +9% for primary and +4% for secondary rays in 4-ary BVH), and it is not suitable for 8-ary BVH (-2% in secondary rays). This modest improvement may be due to the perfectly diffuse (Lambertian) reflection used in all scenes.

In contrast, the quantization of wide BVHs leads to a very significant performance increase, up to 58% compared to the uncompressed counterpart. Potentially, this could be improved further by employing stack compression, other methods to optimize traversal order and computing quantized ray-box intersections more efficiently.

The main difficulty during implementation was the unpredictable register allocator: spilling to shared memory and allocating an excessive number of registers significantly reduced BVH traversal performance. From our point of view, it would be highly beneficial to introduce a Vulkan extension that provides more control over the register allocator.

The evaluated Slang shading language shows performance of secondary rays nearly comparable to binary BVH (+1–2.5%), but suffers from the register allocator limitations (and therefore up to -14% for secondary rays and -28% for primary rays). However, the implemented shaders are more readable and maintainable due to several practical language features. Unfortunately, Slang is still a rapidly evolving project and lacks maturity in some areas.

As a possible future improvement, it would be worthwhile to implement a wide BVH contraction method that processes internal nodes and leaves simultaneously, test view-dependent BVHs in scenes with more specular reflectance (which would require adding configurable materials to Orchard), and, most importantly, find a solution for shader resource allocation, which remains an issue for some shaders.

References

1. KAJIYA, James T. The rendering equation. *SIGGRAPH Comput. Graph.* 1986, vol. 20, no. 4, pp. 143–150. ISSN 0097-8930. Available from DOI: 10.1145/15886.15902.
2. MEISTER, Daniel; OGAKI, Shinji; BENTHIN, Carsten; DOYLE, Michael J.; GUTHE, Michael; BITTNER, Jiří. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum.* 2021, vol. 40, no. 2, pp. 683–712. Available from DOI: 10.1111/cgf.142662.
3. WHITTED, Turner. An improved illumination model for shaded display. *Commun. ACM.* 1980, vol. 23, no. 6, pp. 343–349. ISSN 0001-0782. Available from DOI: 10.1145/358876.358882.
4. CLARK, James H. Hierarchical geometric models for visible surface algorithms. *Commun. ACM.* 1976, vol. 19, no. 10, pp. 547–554. ISSN 0001-0782. Available from DOI: 10.1145/360349.360354.
5. ERICSON, Christer. *Real-Time Collision Detection*. CRC Press, 2004. Available from DOI: 10.1201/b14581.
6. MACDONALD, David J.; BOOTH, Kellogg S. Heuristics for ray tracing using space subdivision. *Vis. Comput.* 1990, vol. 6, no. 3, pp. 153–166. ISSN 0178-2789. Available from DOI: 10.1007/BF01911006.
7. ADVANCED MICRO DEVICES, INC. *Hardware implementation* [HIP Documentation] [online]. [N.d.]. [visited on 2025-05-16]. Available from: https://rocm.docs.amd.com/projects/HIP/en/latest/understand/hardware_implementation.html.
8. NVIDIA CORPORATION. *CUDA C++ Programming Guide* [online]. [N.d.]. [visited on 2025-05-16]. Available from: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
9. OVERVOORDE, Alexander. *Compute Shader* [Khronos Vulkan Tutorial] [online]. [N.d.]. [visited on 2025-05-16]. Available from: https://docs.vulkan.org/tutorial/latest/11_Compute_Shader.html.
10. KHRONOS VULKAN WORKING GROUP. *Vulkan® 1.4.315 - A Specification* [online]. 2025. [visited on 2025-05-16]. Available from: <https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html>.
11. LAUTERBACH, C.; GARLAND, M.; SENGUPTA, S.; LUEBKE, D.; MANOCHA, D. Fast BVH Construction on GPUs. *Computer Graphics Forum.* 2009, vol. 28, no. 2, pp. 375–384. Available from DOI: 10.1111/j.1467-8659.2009.01377.x.
12. APETREI, Ciprian. Fast and Simple Agglomerative LBVH Construction. In: *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association, 2014. ISBN 978-3-905674-70-5. Available from DOI: 10.2312/cgvc.20141206.
13. MEISTER, Daniel; BITTNER, Jiří. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics.* 2018, vol. 24, no. 3, pp. 1345–1353.
14. BENTHIN, Carsten; DRABINSKI, Radosław; TESSARI, Lorenzo; DITTEBRANDT, Addis. PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. *Proc. ACM Comput. Graph. Interact. Tech.* 2022, vol. 5, no. 3. Available from DOI: 10.1145/3543867.

15. BENTHIN, Carsten; MEISTER, Daniel; BARCZAK, Joshua; MEHALWAL, Rohan; TSAKOK, John; KENSLER, Andrew. H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy construction. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*. 2024, vol. 7, no. 3, pp. 1–14. Available from DOI: 10.1145/3675377.
16. KARRAS, Tero; AILA, Timo. Fast parallel construction of high-quality bounding volume hierarchies. In: *Proceedings of the 5th High-Performance Graphics Conference*. Anaheim, California: Association for Computing Machinery, 2013, pp. 89–99. HPG '13. ISBN 9781450321358. Available from DOI: 10.1145/2492045.2492055.
17. GU, Yan; HE, Yong; BLELLOCH, Guy E. Ray Specialized Contraction on Bounding Volume Hierarchies. *Computer Graphics Forum*. 2015, vol. 34, no. 7, pp. 309–318. Available from DOI: 10.1111/cgf.12769.
18. YLITIE, Henri; KARRAS, Tero; LAINE, Samuli. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In: *Proceedings of High Performance Graphics*. Los Angeles, California: Association for Computing Machinery, 2017. HPG '17. ISBN 9781450351010. Available from DOI: 10.1145/3105762.3105773.
19. AILA, Timo; LAINE, Samuli. *Understanding the efficiency of ray traversal on GPUs*. ACM, 2009. Available from DOI: 10.1145/1572769.1572792.
20. GUPTA, Kshitij; STUART, Jeff A.; OWENS, John D. A study of Persistent Threads style GPU programming for GPGPU workloads. In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–14. Available from DOI: 10.1109/InPar.2012.6339596.
21. KNUTH, Donald E. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN 0201896850.
22. GARANZHA, Kirill; LOOP, Charles. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*. 2010, vol. 29, no. 2, pp. 289–298. Available from DOI: 10.1111/j.1467-8659.2009.01598.x.
23. BERTSEKAS, Dimitri P. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*. 1992, vol. 1, no. 1, pp. 7–66. Available from DOI: 10.1007/bf00247653.
24. OGAKI, Shinji; DEROUET-JOURDAN, Alexandre. An N-ary BVH Child Node Sorting Technique for Occlusion Tests. *Journal of Computer Graphics Techniques (JCGT)*. 2016, vol. 5, no. 2, pp. 22–37. ISSN 2331-7418. Available also from: <http://jcgt.org/published/0005/02/02/>.
25. KESSENICH, John. *An Introduction to SPIR-V: A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels*. 2015. White Paper. LunarG. Available also from: <https://registry.khronos.org/SPIR-V/papers/WhitePaper.pdf>.
26. LEESE, Graeme; KESSENICH, John; BALDWIN, Dave; ROST, Randi. *The OpenGL® Shading Language, Version 4.60.8*. The Khronos Group Inc., 2023. Available also from: <https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>.

27. HE, Yong. *Slang – A Shader Compilation System for Extensible, Real-Time Shading*. 2018.
Available also from:
http://graphics.cs.cmu.edu/projects/renderergenerator/yong_he_thesis.pdf.
PhD thesis. Carnegie Mellon University.
28. THE KHRONOS GROUP INC. *Khronos Group Launches Slang Initiative, Hosting Open Source Compiler Contributed by NVIDIA* [online]. 2024-11-21. [visited on 2025-01-19].
Available from: <https://khr.io/17f>.
29. SLANG CONTRIBUTORS. *Slang User's Guide* [online]. [visited on 2025-01-19].
Available from: <https://shader-slang.org/slang/user-guide/>.
30. *Issues by user 'cezneluk' in shader-slang/slang* [online]. 2024. [visited on 2025-01-19].
Available from:
<https://github.com/shader-slang/slang/issues?q=is%3Aissue%20author%3Acezneluk>.
31. KÁČETIK, Martin; BITTNER, Jiří. SAH-Optimized k-DOP Hierarchies for Ray Tracing. *Proc. ACM Comput. Graph. Interact. Tech.* 2024, vol. 7, no. 3.
Available from doi: 10.1145/3675391.
32. LAINE, Samuli; KARRAS, Tero; AILA, Timo.
Megakernels considered harmful: wavefront path tracing on GPUs. In:
Proceedings of the 5th High-Performance Graphics Conference.
Anaheim, California: Association for Computing Machinery, 2013, pp. 137–143. HPG '13.
ISBN 9781450321358. Available from doi: 10.1145/2492045.2492060.
33. NVIDIA CORPORATION. *NVIDIA Management Library*.
Available also from: <https://developer.nvidia.com/management-library-nvml>.
34. D-BUS CONTRIBUTORS. *D-Bus* [online]. [visited on 2025-05-23].
Available from: <https://www.freedesktop.org/wiki/Software/dbus/>.
35. NVIDIA CORPORATION. *CUDA GPU Occupancy Calculator spreadsheet* [online].
2011. [visited on 2025-05-16]. Available from: https://developer.download.nvidia.com/compute/cuda/4_0/sdk/docs/CUDA_Occupancy_Calculator.xls.
36. NVIDIA CORPORATION. *NVIDIA Nsight Compute*. 2018–2024.
Available also from: <https://developer.nvidia.com/nsight-compute>.
37. THE KHRONOS GROUP INC. *VkPhysicalDeviceLimits(3) Manual Page* [online].
2014–2025. [visited on 2025-05-14]. Available from: <https://registry.khronos.org/vulkan/specs/latest/man/html/VkPhysicalDeviceLimits.html>.
38. MICIKEVICIUS, Paulius. *Local Memory and Register Spilling* [online].
2011. [visited on 2025-05-11]. Available from:
https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.
39. NVIDIA CORPORATION. *NVIDIA Nsight Graphics*. 2018–2025.
Available also from: <https://developer.nvidia.com/nsight-graphics>.
40. *ForceInline moves instructions before a condition · Issue #6654 · shader-slang/slang* [online].
2025. [visited on 2025-05-14].
Available from: <https://github.com/shader-slang/slang/issues/6654>.
41. BAKER, Alan. *VK_KHR_shader_maximal_reconvergence(3) Manual Page* [online].
2021. [visited on 2025-05-14]. Available from: https://registry.khronos.org/vulkan/specs/latest/man/html/VK_KHR_shader_maximal_reconvergence.html.
42. AMAZON LUMBERYARD.
Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). 2017.
Available also from: <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>.

43. MCGUIRE, Morgan. *Computer Graphics Archive*. 2017.
Available also from: <https://casual-effects.com/data>.
44. TRAN, Robin. *Blender 2.91 splash screen – Red Autumn Forest* [online].
2020. [visited on 2025-01-15].
Available from: <https://cloud.blender.org/p/gallery/5fbd186ec57d586577c57417>.
45. LYNXSDESIGN. *Blender 4.1 splash screen – Lynxsdesign* [online].
2024. [visited on 2025-01-15]. Available from:
<https://www.blender.org/download/demo/splash/blender-4.1-splash.blend>.

List of Figures

1	Schematic visualization of Whitted Ray Tracing and Path Tracing.	1
2	An example of a BVH tree.	2
3	Usual types of bounding volumes.	3
4	Example of the Vulkan execution model.	4
5	Example of the contraction of an internal BVH node.	5
6	Example of conversion from a binary BVH tree to a 4-ary wide BVH tree using the HPLOC algorithm.	9
7	Compressed 8-ary BVH node by Ylitie et al. ^[18]	10
8	Optimal sorting networks for various numbers of elements.	12
9	Memory layout of a binary BVH node, 56 bytes in total.	17
10	Diagram of wavefront shaders and their order.	17
11	Memory layout of an uncompressed 4-ary BVH node, 120 bytes in total.	19
12	Encoding of a child index, representing either an internal node or a leaf which containing triangles.	19
13	Memory layout of a quantized 4-ary BVH node, 52 bytes in total.	20
14	Encoding of quantization step exponents together with the counts of children and leaves into a 4-byte integer.	20
15	Encoding of a quantized bounding volume into one and a half 4-byte integers.	20
16	Example of a Git commit showing a shader comparison text.	23
17	Comparison of two profile captures of the identical binary single kernel traversal shader.	27
18	Analysis of active registers in the quantized 8-ary BVH Slang traversal shader.	28
19	Relative MRps performance of Slang shaders before and after the instruction hoisting fix.	29
20	Relative MRps performance of Slang shaders with and without the ray-triangle intersection modification.	31
21	The relative MRps performance of Slang shaders with and without workgroup barrier.	32
22	Example view for each testing scene.	33
23	Relation between the CBTC heuristic threshold t and the relative MRps performance of the resulting BVH.	36
24	Relation between the depth limit d of sample rays and relative MRps performance on the resulting BVH.	37
25	Relation between subsampling ratio s and relative MRps performance on the resulting BVH.	38
26	Benchmark results for all scenes.	40
27	Results of uncompressed BVH variants for each scene.	52
28	Relative MRps performance of quantized BVHs compared to their uncompressed variant.	52
29	Relative MRps performance of uncompressed view-dependent BVHs compared to their uncompressed view-independent variant.	53

30	Relative MRps performance of quantized view-dependent BVHs compared to their quantized view-independent variant.	53
----	--	----

List of Tables

1	Shader constraints and parameters of the evaluated GPU.	25
2	Comparison of shader resource allocation.	26
3	Relative MRps comparison between the separated and single kernel traversal shaders.	27
4	Overview of the selected workgroup sizes across all shader variants.	34
5	Overview of the selected SAH traversal cost c_t across all shader variants.	35
6	Average relative MRps of the scenes using varying c_t parameters for leaf node compaction.	35
7	The relative primary rays MRps performance of GLSL shaders with different workgroup sizes.	49
8	The relative secondary rays MRps performance of GLSL shaders with different workgroup sizes.	49
9	The relative primary rays MRps performance of GLSL shaders with workgroup barrier at different workgroup sizes.	50
10	The relative secondary rays MRps performance of Slang shaders with workgroup barrier with different workgroup sizes.	50
11	The relative primary rays MRps performance of Slang shaders without workgroup barrier with different workgroup sizes.	51
12	The relative secondary rays MRps performance of Slang shaders without workgroup barrier with different workgroup sizes.	51
13	Results of all shader variants for the bistro_int scene.	54
14	Results of all shader variants for the bistro_ext scene.	55
15	Results of all shader variants for the lynxsdesign scene.	56
16	Results of all shader variants for the san_miguel scene.	57
17	Results of all shader variants for the red_autumn_forest scene.	58

List of Listings

1	Pseudocode of the BVH traversal algorithm with two separate loops.	11
2	Pseudocode of BVH traversal while-while algorithm with persistent threads and speculative traversal.	12
3	The definition of a uncompressed wide BVH node.	19
4	The definition of a quantized wide BVH node.	21
5	Comparison of the SPIR-V assembly of the ray-triangle intersection code before and after applying the instruction hoisting fix.	30

Appendices

A MRps Performance for Various Workgroup Configuration

Primary rays GLSL		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary	min	0.87	0.88	0.88	0.88	0.88	1.00	0.88	0.93	0.96
	avg	0.95	0.95	0.95	0.95	0.96	1.00	0.96	0.97	0.98
Wide4	min	0.99	0.98	0.99	0.82	0.81	1.00	0.81	0.91	0.96
	avg	0.99	0.99	0.99	0.93	0.93	1.00	0.93	0.92	0.98
Wide4 Quantized	min	0.96	0.95	0.96	1.00	0.98	0.97	0.80	0.95	0.97
	avg	0.96	0.96	0.96	1.00	1.00	0.97	0.83	0.98	0.99
Wide6	min	1.00	1.00	1.00	0.97	0.88	1.00	0.79	0.91	0.95
	avg	1.00	1.00	1.00	0.98	0.91	1.00	0.82	0.94	0.98
Wide6 Quantized	min	0.97	0.96	0.97	0.99	0.97	1.00	0.90	0.97	0.99
	avg	0.98	0.98	0.99	1.00	0.99	1.00	0.93	1.00	1.01
Wide8	min	1.00	1.00	0.97	1.00	0.97	0.98	0.89	0.94	0.98
	avg	1.00	1.00	0.97	1.00	0.98	1.00	0.90	0.95	0.98
Wide8 Quantized	min	0.96	0.96	0.98	0.98	0.98	1.00	0.83	0.97	0.98
	avg	0.97	0.97	0.99	0.99	1.00	1.00	0.87	0.99	0.99

Table 7: The relative primary rays MRps performance of GLSL shaders with different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and highlighted in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

Secondary rays GLSL		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary	min	0.89	0.89	0.89	0.89	0.89	0.95	0.89	0.97	1.00
	avg	0.91	0.91	0.91	0.91	0.91	0.97	0.91	1.04	1.00
Wide4	min	1.00	1.00	1.00	0.77	0.76	0.98	0.73	0.90	0.96
	avg	1.00	1.00	1.00	0.84	0.84	0.99	0.82	1.02	1.02
Wide4 Quantized	min	0.87	0.87	0.87	1.00	0.95	0.86	0.68	0.88	0.94
	avg	0.93	0.93	0.93	1.00	0.99	0.93	0.79	0.96	0.98
Wide6	min	0.92	0.92	0.92	0.93	0.80	0.92	0.78	0.96	1.00
	avg	0.94	0.94	0.94	0.95	0.87	0.94	0.87	1.01	1.00
Wide6 Quantized	min	0.92	0.92	0.94	1.00	0.96	0.94	0.80	0.90	0.96
	avg	0.97	0.97	0.98	1.00	0.97	0.98	0.87	0.95	0.98
Wide8	min	1.00	1.00	0.97	1.00	0.97	0.93	0.87	0.94	0.97
	avg	1.00	1.00	1.00	1.00	1.00	0.96	0.98	1.01	1.00
Wide8 Quantized	min	0.80	0.80	0.85	1.00	0.97	0.85	0.64	0.92	0.98
	avg	0.90	0.90	0.94	1.00	1.00	0.94	0.71	0.97	1.00

Table 8: The relative secondary rays MRps performance of GLSL shaders with different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and highlighted in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

Primary rays Slang with barrier		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary Slang	min	1.00	0.93	0.88	0.81	0.75	0.81	0.61	0.66	0.70
	avg	1.00	0.95	0.90	0.83	0.78	0.83	0.64	0.68	0.72
Wide4 Slang	min	1.00	0.93	0.84	0.84	0.78	0.83	0.64	0.69	0.72
	avg	1.00	0.94	0.86	0.86	0.81	0.85	0.67	0.72	0.75
Wide4 Slang Quantized	min	1.00	0.94	0.89	0.75	0.84	0.59	0.70	0.75	0.79
	avg	1.00	0.95	0.90	0.78	0.86	0.62	0.73	0.77	0.80
Wide6 Slang	min	1.00	0.94	0.85	0.88	0.80	0.61	0.68	0.73	0.75
	avg	1.00	0.95	0.88	0.90	0.83	0.64	0.70	0.74	0.77
Wide6 Slang Quantized	min	1.00	0.93	0.83	0.81	0.89	0.64	0.76	0.79	0.82
	avg	1.00	0.94	0.85	0.82	0.90	0.66	0.77	0.80	0.83
Wide8 Slang	min	1.00	0.93	0.89	0.73	0.83	0.62	0.68	0.73	0.78
	avg	1.00	0.94	0.91	0.76	0.86	0.65	0.70	0.74	0.79
Wide8 Slang Quantized	min	1.00	0.94	0.91	0.80	0.87	0.71	0.76	0.79	0.81
	avg	1.00	0.95	0.92	0.82	0.88	0.73	0.77	0.80	0.83

Table 9: The relative primary rays MRps performance of GLSL shaders with workgroup barrier at different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and shown in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

Secondary rays Slang with barrier		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary Slang	min	1.00	0.95	0.90	0.83	0.76	0.82	0.60	0.65	0.69
	avg	1.00	1.02	1.02	1.01	0.99	0.99	0.90	0.93	0.94
Wide4 Slang	min	1.00	0.94	0.84	0.85	0.78	0.83	0.61	0.66	0.70
	avg	1.00	1.00	0.98	0.98	0.95	0.94	0.85	0.88	0.90
Wide4 Slang Quantized	min	1.00	0.95	0.90	0.75	0.86	0.58	0.71	0.76	0.81
	avg	1.00	0.96	0.92	0.78	0.88	0.61	0.74	0.80	0.84
Wide6 Slang	min	1.00	0.95	0.86	0.87	0.80	0.58	0.65	0.69	0.73
	avg	1.00	1.00	0.99	0.98	0.96	0.79	0.86	0.89	0.90
Wide6 Slang Quantized	min	1.00	0.94	0.83	0.80	0.91	0.63	0.75	0.80	0.84
	avg	1.00	0.95	0.84	0.82	0.92	0.65	0.78	0.82	0.86
Wide8 Slang	min	1.00	0.93	0.88	0.69	0.81	0.56	0.63	0.67	0.72
	avg	1.00	0.99	0.98	0.90	0.96	0.80	0.87	0.90	0.92
Wide8 Slang Quantized	min	1.00	0.95	0.94	0.80	0.89	0.70	0.77	0.81	0.83
	avg	1.00	0.96	0.94	0.82	0.90	0.72	0.78	0.82	0.85

Table 10: The relative secondary rays MRps performance of Slang shaders with workgroup barrier with different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and shown in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

Primary rays Slang w/o barrier		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary Slang	min	0.97	0.97	0.97	0.98	0.99	0.98	0.90	0.96	1.00
	avg	0.98	0.99	0.99	0.99	0.99	0.99	0.91	0.97	1.00
Wide4 Slang	min	0.96	0.96	0.96	0.97	0.96	1.00	0.88	0.94	0.97
	avg	0.97	0.97	0.97	0.98	0.98	1.00	0.91	0.97	0.98
Wide4 Slang Quantized	min	0.94	0.94	0.95	0.91	0.96	0.78	0.92	0.97	1.00
	avg	0.97	0.97	0.97	0.94	0.98	0.82	0.95	0.98	1.00
Wide6 Slang	min	0.96	0.96	0.96	1.00	0.96	0.80	0.89	0.94	0.97
	avg	0.97	0.97	0.96	1.00	0.97	0.86	0.93	0.97	0.99
Wide6 Slang Quantized	min	0.94	0.94	0.92	0.92	0.97	0.80	0.94	0.96	1.00
	avg	0.95	0.95	0.94	0.94	0.98	0.83	0.96	0.97	1.00
Wide8 Slang	min	0.94	0.94	0.96	0.89	0.97	0.70	0.90	0.95	1.00
	avg	0.95	0.95	0.98	0.91	0.99	0.73	0.92	0.95	1.00
Wide8 Slang Quantized	min	0.96	0.95	0.94	0.92	0.96	0.88	0.94	0.97	1.00
	avg	0.97	0.97	0.96	0.94	0.97	0.91	0.95	0.98	1.00

Table 11: The relative primary rays MRps performance of Slang shaders without workgroup barrier with different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and shown in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

Secondary rays Slang w/o barrier		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary Slang	min	0.95	0.95	0.95	0.98	1.00	0.93	0.85	0.94	1.00
	avg	1.00	1.00	1.00	1.00	1.00	0.99	0.92	0.97	1.00
Wide4 Slang	min	0.98	0.98	1.00	0.98	1.00	0.92	0.87	0.95	1.00
	avg	1.01	1.01	1.00	1.01	1.00	1.00	0.92	0.98	1.00
Wide4 Slang Quantized	min	0.98	0.98	0.98	0.87	0.99	0.73	0.88	0.96	1.00
	avg	0.98	0.98	0.99	0.89	0.99	0.75	0.89	0.97	1.00
Wide6 Slang	min	0.98	0.98	0.99	0.98	0.99	0.77	0.89	0.96	1.00
	avg	1.01	1.01	1.00	1.01	1.00	0.83	0.93	0.99	1.00
Wide6 Slang Quantized	min	0.94	0.94	0.88	0.88	1.00	0.73	0.89	0.95	1.00
	avg	0.95	0.95	0.89	0.90	1.00	0.76	0.91	0.97	1.00
Wide8 Slang	min	0.94	0.94	0.96	0.89	0.97	0.70	0.90	0.95	1.00
	avg	0.95	0.95	0.98	0.91	0.99	0.73	0.92	0.95	1.00
Wide8 Slang Quantized	min	0.95	0.95	0.98	0.87	0.98	0.79	0.89	0.95	1.00
	avg	0.96	0.96	0.98	0.89	0.99	0.82	0.90	0.96	1.00

Table 12: The relative secondary rays MRps performance of Slang shaders without workgroup barrier with different workgroup sizes. Values are averaged across testing scenes. The best workgroup size for each variant (selected primarily based on the minimum value) is chosen as the reference and shown in bold black text. Sizes where the relative MRps falls below 0.9 are shown in bold red text.

B Per-Scene Results

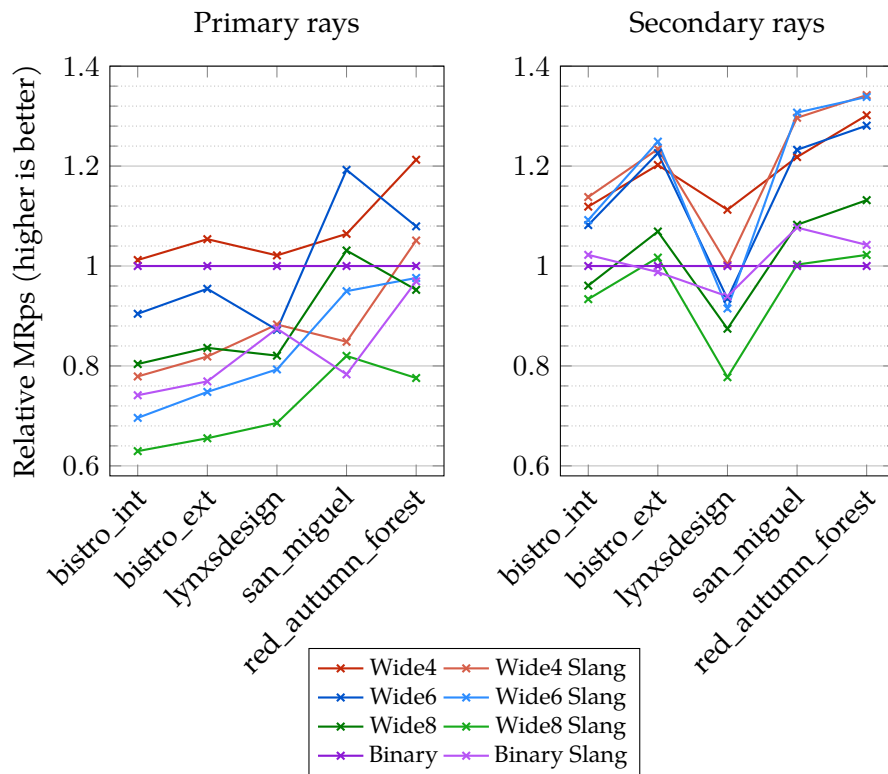


Figure 27: Results of uncompressed BVH variants for each scene. All values are relative to the Binary variant.

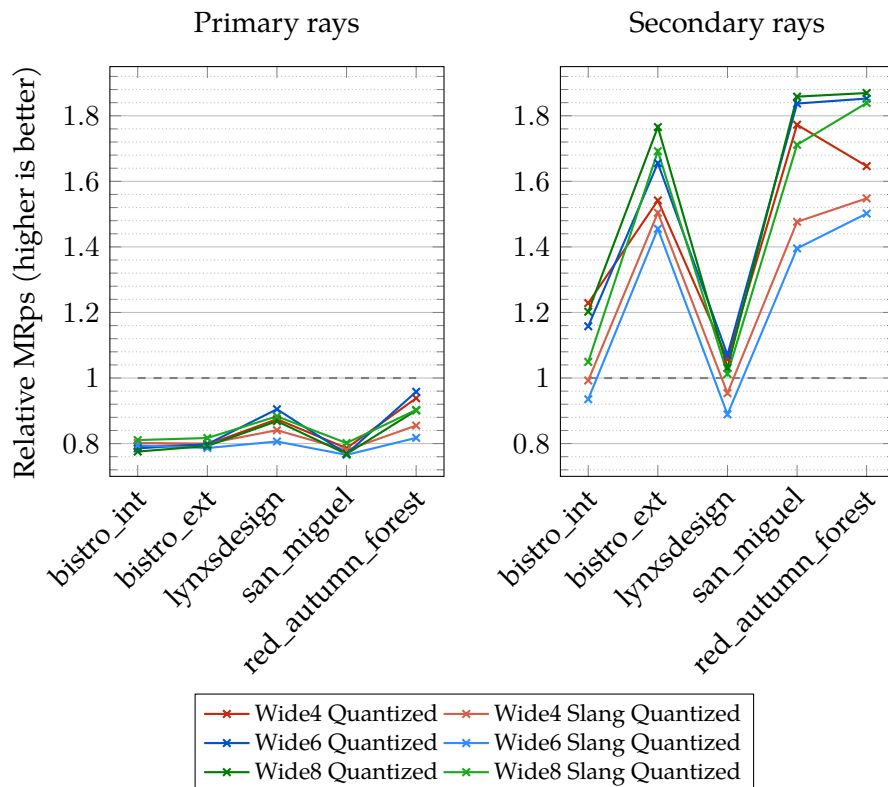


Figure 28: Relative MRps performance of quantized BVHs compared to their uncompressed variant.

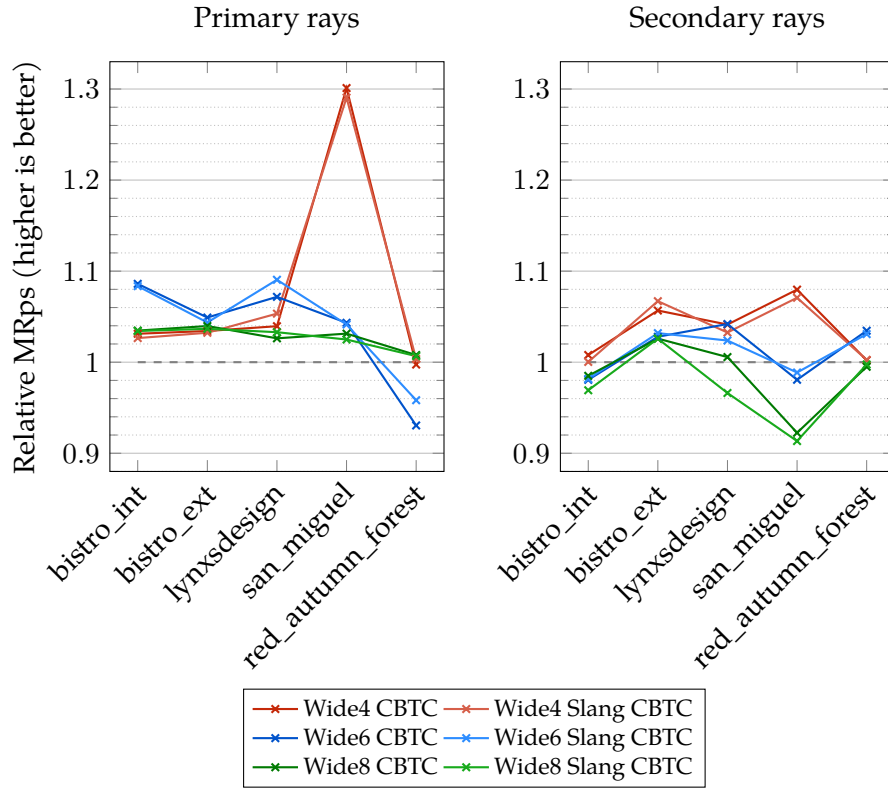


Figure 29: Relative MRps performance of uncompressed view-dependent BVHs compared to their uncompressed view-independent variant.

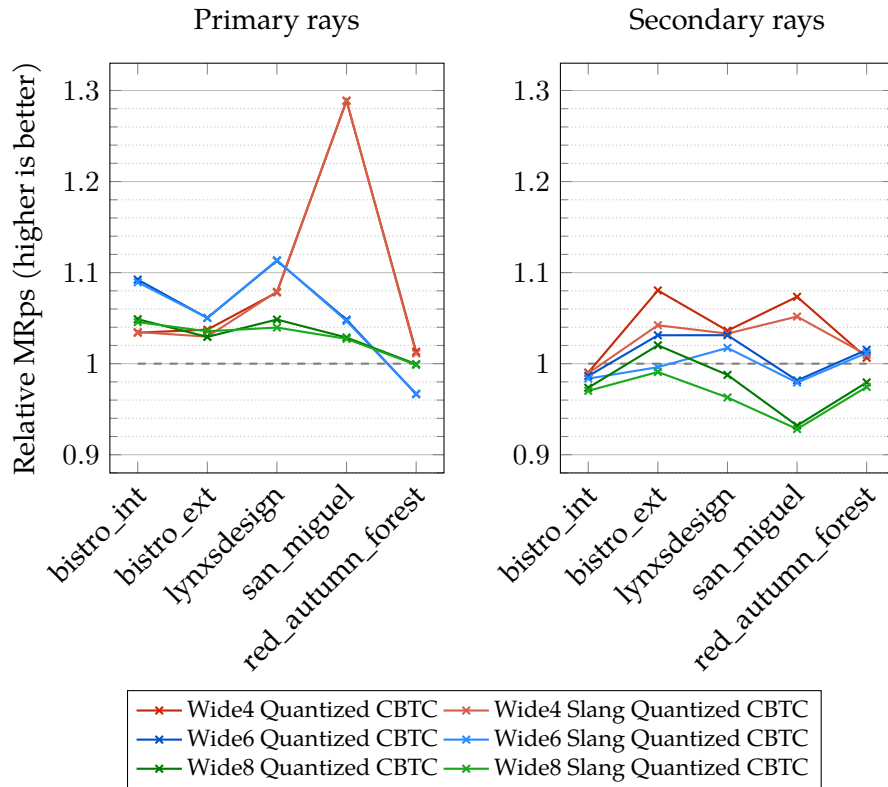


Figure 30: Relative MRps performance of quantized view-dependent BVHs compared to their quantized view-independent variant.


 bistro_int (1.04 Mtris)	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)	Tested nodes per ray	Tested BVs per ray	Tested triangles per ray
Binary	11.6 (1.00)	2.0	2.1	1772 (1.00)	226 (1.00)	59.9 (1.00)	119.8 (1.00)	66.4 (1.00)
Binary Slang	11.6 (1.00)	2.0		1314 (0.74)	231 (1.02)	59.8 (1.00)	119.6 (1.00)	66.2 (1.00)
Wide4	11.9 (1.02)	3.1		1794 (1.01)	253 (1.12)	29.7 (0.50)	114.6 (0.96)	63.6 (0.96)
Wide4 CBTC	2.2 + 12.6 (1.27)	3.2		1850 (1.04)	255 (1.13)	28.7 (0.48)	111.3 (0.93)	64.5 (0.97)
Wide4 Quantized	12.4 (1.06)	3.1		1420 (0.80)	311 (1.37)	30.2 (0.50)	116.7 (0.97)	64.1 (0.97)
Wide4 Quantized CBTC	2.2 + 13.2 (1.32)	3.2		1468 (0.83)	308 (1.36)	29.2 (0.49)	113.5 (0.95)	65.0 (0.98)
Wide4 Slang	11.9 (1.02)	3.1		1381 (0.78)	258 (1.14)	29.6 (0.49)	114.5 (0.96)	63.5 (0.96)
Wide4 Slang CBTC	2.3 + 12.6 (1.28)	3.2		1417 (0.80)	258 (1.14)	28.6 (0.48)	111.2 (0.93)	64.3 (0.97)
Wide4 Slang Quantized	12.4 (1.06)	3.1		1107 (0.62)	256 (1.13)	30.2 (0.50)	116.6 (0.97)	64.0 (0.96)
Wide4 Slang Quantized CBTC	2.3 + 13.2 (1.33)	3.2		1145 (0.65)	253 (1.12)	29.2 (0.49)	113.3 (0.95)	64.9 (0.98)
Wide6	11.9 (1.02)	3.7	2.5	1603 (0.90)	245 (1.08)	23.0 (0.38)	129.4 (1.08)	65.3 (0.98)
Wide6 CBTC	2.2 + 12.5 (1.26)	4.0		1741 (0.98)	240 (1.06)	22.2 (0.37)	126.5 (1.06)	66.2 (1.00)
Wide6 Quantized	12.8 (1.10)	3.7		1259 (0.71)	283 (1.25)	23.5 (0.39)	132.2 (1.10)	66.2 (1.00)
Wide6 Quantized CBTC	2.2 + 13.1 (1.32)	4.0		1375 (0.78)	280 (1.24)	22.7 (0.38)	129.7 (1.08)	67.0 (1.01)
Wide6 Slang	11.9 (1.02)	3.7		1234 (0.70)	247 (1.09)	23.0 (0.38)	129.3 (1.08)	65.2 (0.98)
Wide6 Slang CBTC	2.2 + 12.5 (1.26)	4.0		1337 (0.75)	243 (1.07)	22.1 (0.37)	126.4 (1.06)	66.1 (1.00)
Wide6 Slang Quantized	12.3 (1.05)	3.8	3.3	980 (0.55)	231 (1.02)	23.3 (0.39)	130.8 (1.09)	67.4 (1.02)
Wide6 Slang Quantized CBTC	2.2 + 13.1 (1.31)	4.0		1068 (0.60)	227 (1.00)	22.5 (0.37)	128.2 (1.07)	68.2 (1.03)
Wide8	11.8 (1.01)	4.3		1425 (0.80)	217 (0.96)	19.5 (0.33)	143.9 (1.20)	67.5 (1.02)
Wide8 CBTC	2.3 + 12.4 (1.27)	4.7		1474 (0.83)	214 (0.95)	19.3 (0.32)	144.3 (1.20)	67.9 (1.02)
Wide8 Quantized	12.9 (1.11)	4.2	2.5	1105 (0.62)	261 (1.16)	20.2 (0.34)	149.3 (1.25)	67.1 (1.01)
Wide8 Quantized CBTC	2.2 + 13.4 (1.34)	4.7		1159 (0.65)	254 (1.12)	20.1 (0.33)	149.9 (1.25)	67.6 (1.02)
Wide8 Slang	11.8 (1.01)	4.3	3.3	1116 (0.63)	211 (0.93)	19.5 (0.32)	143.7 (1.20)	67.4 (1.02)
Wide8 Slang CBTC	2.4 + 12.4 (1.27)	4.7		1154 (0.65)	205 (0.90)	19.3 (0.32)	144.2 (1.20)	67.8 (1.02)
Wide8 Slang Quantized	12.3 (1.06)	4.3		905 (0.51)	222 (0.98)	20.0 (0.33)	147.6 (1.23)	68.3 (1.03)
Wide8 Slang Quantized CBTC	2.3 + 13.1 (1.33)	4.7		946 (0.53)	215 (0.95)	19.8 (0.33)	148.0 (1.24)	68.8 (1.04)

Table 13: Results of all shader variants for the bistro_int scene. Values in parentheses are relative to the Binary variant. The optimal value within each category is highlighted in bold. For the build time of view-dependent BVHs, the first value represents the elapsed time for tracing sample rays, and the second value is the total build time for both BVHs.


 bistro_ext (2.83 Mtris)	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)	Tested nodes per ray	Tested BVs per ray	Tested triangles per ray
Binary	19.8 (1.00)	2.0	2.1	1013 (1.00)	122 (1.00)	79.8 (1.00)	159.6 (1.00)	38.5 (1.00)
Binary Slang	19.8 (1.00)	2.0		779 (0.77)	120 (0.99)	79.7 (1.00)	159.3 (1.00)	38.3 (1.00)
Wide4	20.7 (1.05)	3.1		1067 (1.05)	147 (1.20)	40.1 (0.50)	155.6 (0.97)	40.2 (1.05)
Wide4 CBTC	2.7 + 22.2 (1.26)	3.2		1104 (1.09)	155 (1.27)	38.7 (0.49)	150.9 (0.95)	39.6 (1.03)
Wide4 Quantized	21.6 (1.09)	3.1		851 (0.84)	226 (1.85)	40.9 (0.51)	158.6 (0.99)	40.8 (1.06)
Wide4 Quantized CBTC	2.7 + 24.1 (1.36)	3.2		882 (0.87)	244 (2.00)	39.5 (0.50)	154.0 (0.96)	40.2 (1.05)
Wide4 Slang	20.4 (1.03)	3.1		830 (0.82)	150 (1.23)	40.0 (0.50)	155.3 (0.97)	40.1 (1.04)
Wide4 Slang CBTC	2.8 + 22.3 (1.27)	3.2		856 (0.85)	161 (1.32)	38.7 (0.48)	150.7 (0.94)	39.5 (1.03)
Wide4 Slang Quantized	21.9 (1.11)	3.1		664 (0.66)	226 (1.86)	40.8 (0.51)	158.4 (0.99)	40.7 (1.06)
Wide4 Slang Quantized CBTC	2.7 + 24.2 (1.36)	3.2		683 (0.67)	236 (1.93)	39.5 (0.49)	153.8 (0.96)	40.1 (1.04)
Wide6	20.0 (1.01)	3.8	2.6	967 (0.95)	149 (1.23)	30.8 (0.39)	175.6 (1.10)	41.7 (1.08)
Wide6 CBTC	2.6 + 22.1 (1.25)	4.0		1014 (1.00)	154 (1.26)	29.6 (0.37)	170.3 (1.07)	42.6 (1.11)
Wide6 Quantized	22.1 (1.12)	3.8		772 (0.76)	247 (2.03)	31.5 (0.40)	179.6 (1.13)	42.6 (1.11)
Wide6 Quantized CBTC	2.6 + 23.8 (1.34)	4.0		811 (0.80)	255 (2.09)	30.4 (0.38)	174.6 (1.09)	43.5 (1.13)
Wide6 Slang	20.0 (1.01)	3.8		758 (0.75)	152 (1.25)	30.8 (0.39)	175.3 (1.10)	41.6 (1.08)
Wide6 Slang CBTC	2.7 + 22.1 (1.25)	4.0		791 (0.78)	157 (1.29)	29.6 (0.37)	170.0 (1.07)	42.5 (1.11)
Wide6 Slang Quantized	21.1 (1.07)	3.9	3.3	596 (0.59)	222 (1.82)	31.0 (0.39)	176.8 (1.11)	45.1 (1.17)
Wide6 Slang Quantized CBTC	2.6 + 23.0 (1.30)	4.1		626 (0.62)	221 (1.81)	29.9 (0.37)	171.7 (1.08)	46.1 (1.20)
Wide8	19.7 (1.00)	4.4		847 (0.84)	130 (1.07)	26.5 (0.33)	197.6 (1.24)	44.9 (1.17)
Wide8 CBTC	2.6 + 21.5 (1.22)	4.8		881 (0.87)	134 (1.10)	25.7 (0.32)	194.2 (1.22)	45.7 (1.19)
Wide8 Quantized	21.8 (1.11)	4.3	2.6	671 (0.66)	230 (1.89)	27.7 (0.35)	206.0 (1.29)	43.3 (1.13)
Wide8 Quantized CBTC	2.6 + 24.1 (1.35)	4.7		691 (0.68)	235 (1.93)	27.0 (0.34)	203.4 (1.27)	44.3 (1.15)
Wide8 Slang	20.1 (1.01)	4.4	3.3	664 (0.66)	124 (1.02)	26.5 (0.33)	197.4 (1.24)	44.8 (1.17)
Wide8 Slang CBTC	2.7 + 21.5 (1.22)	4.8		688 (0.68)	127 (1.04)	25.6 (0.32)	194.0 (1.22)	45.6 (1.19)
Wide8 Slang Quantized	22.1 (1.12)	4.4		542 (0.54)	210 (1.72)	27.2 (0.34)	202.8 (1.27)	45.8 (1.19)
Wide8 Slang Quantized CBTC	2.6 + 23.2 (1.31)	4.8		562 (0.55)	208 (1.70)	26.5 (0.33)	200.1 (1.25)	46.9 (1.22)

Table 14: Results of all shader variants for the bistro_ext scene. Values in parentheses are relative to the Binary variant. The optimal value within each category is highlighted in bold. For the build time of view-dependent BVHs, the first value represents the elapsed time for tracing sample rays, and the second value is the total build time for both BVHs.


<div>lynxsdesign (8.20 Mtris)</div> 	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)	Tested nodes per ray	Tested BVs per ray	Tested triangles per ray
Binary	38.0 (1.00)	2.0	2.0	2461 (1.00)	960 (1.00)	29.9 (1.00)	59.8 (1.00)	5.4 (1.00)
Binary Slang	38.0 (1.00)	2.0		2153 (0.87)	902 (0.94)	29.9 (1.00)	59.7 (1.00)	5.4 (0.99)
Wide4	39.0 (1.03)	3.0		2513 (1.02)	1068 (1.11)	15.4 (0.51)	60.7 (1.01)	5.3 (0.98)
Wide4 CBTC	1.6 + 45.5 (1.24)	3.1		2612 (1.06)	1113 (1.16)	14.2 (0.48)	56.3 (0.94)	5.3 (0.98)
Wide4 Quantized	43.1 (1.13)	3.0		2198 (0.89)	1131 (1.18)	15.8 (0.53)	62.2 (1.04)	5.5 (1.02)
Wide4 Quantized CBTC	1.6 + 51.0 (1.39)	3.1		2370 (0.96)	1172 (1.22)	14.6 (0.49)	57.7 (0.96)	5.5 (1.01)
Wide4 Slang	38.0 (1.00)	3.0		2173 (0.88)	963 (1.00)	15.4 (0.51)	60.6 (1.01)	5.2 (0.97)
Wide4 Slang CBTC	1.7 + 45.8 (1.25)	3.1		2289 (0.93)	994 (1.04)	14.2 (0.48)	56.2 (0.94)	5.2 (0.97)
Wide4 Slang Quantized	43.3 (1.14)	3.0		1827 (0.74)	919 (0.96)	15.8 (0.53)	62.1 (1.04)	5.5 (1.01)
Wide4 Slang Quantized CBTC	1.7 + 51.4 (1.40)	3.1		1971 (0.80)	949 (0.99)	14.6 (0.49)	57.6 (0.96)	5.4 (1.01)
Wide6	37.7 (0.99)	3.6	2.2	2146 (0.87)	898 (0.93)	12.4 (0.41)	72.3 (1.21)	5.5 (1.03)
Wide6 CBTC	1.7 + 45.4 (1.24)	3.9		2300 (0.93)	935 (0.97)	11.1 (0.37)	65.6 (1.10)	5.5 (1.02)
Wide6 Quantized	42.6 (1.12)	3.6		1942 (0.79)	961 (1.00)	12.7 (0.43)	74.4 (1.24)	5.9 (1.09)
Wide6 Quantized CBTC	1.6 + 51.4 (1.40)	3.9		2162 (0.88)	991 (1.03)	11.5 (0.39)	67.8 (1.13)	5.8 (1.07)
Wide6 Slang	38.1 (1.00)	3.6		1952 (0.79)	879 (0.91)	12.4 (0.41)	72.2 (1.21)	5.5 (1.02)
Wide6 Slang CBTC	1.7 + 45.0 (1.23)	3.9		2128 (0.86)	900 (0.94)	11.1 (0.37)	65.5 (1.09)	5.5 (1.01)
Wide6 Slang Quantized	41.3 (1.09)	3.6	2.8	1574 (0.64)	781 (0.81)	12.6 (0.42)	73.9 (1.24)	6.5 (1.20)
Wide6 Slang Quantized CBTC	1.7 + 48.0 (1.31)	3.9		1752 (0.71)	795 (0.83)	11.4 (0.38)	67.5 (1.13)	6.3 (1.17)
Wide8	36.9 (0.97)	4.1		2019 (0.82)	840 (0.87)	10.2 (0.34)	78.8 (1.32)	6.1 (1.13)
Wide8 CBTC	1.7 + 42.4 (1.16)	4.6	2.2	2072 (0.84)	845 (0.88)	9.8 (0.33)	76.8 (1.28)	6.1 (1.13)
Wide8 Quantized	43.1 (1.13)	4.0		1754 (0.71)	866 (0.90)	10.6 (0.35)	81.6 (1.36)	6.0 (1.12)
Wide8 Quantized CBTC	1.6 + 52.7 (1.43)	4.6		1839 (0.75)	855 (0.89)	10.3 (0.34)	80.0 (1.34)	6.0 (1.11)
Wide8 Slang	37.0 (0.98)	4.1	2.8	1688 (0.69)	747 (0.78)	10.1 (0.34)	78.8 (1.32)	6.1 (1.13)
Wide8 Slang CBTC	1.7 + 43.0 (1.18)	4.6		1744 (0.71)	721 (0.75)	9.8 (0.33)	76.8 (1.28)	6.1 (1.13)
Wide8 Slang Quantized	41.7 (1.10)	4.1		1491 (0.61)	756 (0.79)	10.5 (0.35)	81.1 (1.36)	6.7 (1.24)
Wide8 Slang Quantized CBTC	1.7 + 49.8 (1.36)	4.6		1550 (0.63)	728 (0.76)	10.2 (0.34)	79.6 (1.33)	6.6 (1.22)

Table 15: Results of all shader variants for the lynxsdesign scene. Values in parentheses are relative to the Binary variant. The optimal value within each category is highlighted in bold. For the build time of view-dependent BVHs, the first value represents the elapsed time for tracing sample rays, and the second value is the total build time for both BVHs.


<div> <div>san_miguel</div> <div>(9.96 Mtris)</div>  </div>	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)	Tested nodes per ray	Tested BVs per ray	Tested triangles per ray
Binary	53.4 (1.00)	2.0	2.1	880 (1.00)	168 (1.00)	79.3 (1.00)	158.6 (1.00)	13.6 (1.00)
Binary Slang	53.4 (1.00)	2.0		689 (0.78)	181 (1.08)	79.2 (1.00)	158.4 (1.00)	13.6 (1.00)
Wide4	53.3 (1.00)	3.1		936 (1.06)	204 (1.22)	39.3 (0.50)	153.5 (0.97)	12.9 (0.95)
Wide4 CBTC	2.6 + 61.8 (1.21)	3.1		1218 (1.38)	221 (1.32)	36.5 (0.46)	142.8 (0.90)	12.2 (0.90)
Wide4 Quantized	59.4 (1.11)	3.1		736 (0.84)	362 (2.16)	40.3 (0.51)	157.1 (0.99)	13.2 (0.97)
Wide4 Quantized CBTC	2.6 + 68.3 (1.33)	3.1		948 (1.08)	389 (2.32)	37.4 (0.47)	146.2 (0.92)	12.6 (0.92)
Wide4 Slang	53.2 (1.00)	3.1		746 (0.85)	217 (1.30)	39.3 (0.50)	153.3 (0.97)	12.9 (0.95)
Wide4 Slang CBTC	3.0 + 61.8 (1.21)	3.1		963 (1.09)	233 (1.39)	36.4 (0.46)	142.6 (0.90)	12.2 (0.90)
Wide4 Slang Quantized	59.2 (1.11)	3.1		582 (0.66)	321 (1.91)	40.2 (0.51)	157.0 (0.99)	13.2 (0.97)
Wide4 Slang Quantized CBTC	3.0 + 68.7 (1.34)	3.1		750 (0.85)	338 (2.01)	37.3 (0.47)	146.1 (0.92)	12.5 (0.92)
Wide6	52.8 (0.99)	3.7	2.7	1049 (1.19)	207 (1.23)	28.7 (0.36)	164.8 (1.04)	13.4 (0.99)
Wide6 CBTC	2.6 + 58.3 (1.14)	4.0		1094 (1.24)	203 (1.21)	28.3 (0.36)	163.3 (1.03)	13.8 (1.01)
Wide6 Quantized	59.0 (1.11)	3.7		806 (0.92)	380 (2.26)	29.6 (0.37)	169.5 (1.07)	13.8 (1.01)
Wide6 Quantized CBTC	2.6 + 66.3 (1.29)	4.0		845 (0.96)	373 (2.22)	29.1 (0.37)	168.2 (1.06)	14.4 (1.05)
Wide6 Slang	52.0 (0.97)	3.7		835 (0.95)	219 (1.31)	28.7 (0.36)	164.6 (1.04)	13.4 (0.98)
Wide6 Slang CBTC	2.9 + 59.3 (1.17)	4.0		870 (0.99)	217 (1.29)	28.2 (0.36)	163.1 (1.03)	13.8 (1.01)
Wide6 Slang Quantized	56.7 (1.06)	3.8	3.4	639 (0.73)	306 (1.82)	29.1 (0.37)	166.7 (1.05)	16.1 (1.18)
Wide6 Slang Quantized CBTC	2.8 + 63.6 (1.24)	4.0		669 (0.76)	300 (1.79)	28.6 (0.36)	165.2 (1.04)	16.7 (1.23)
Wide8	52.0 (0.97)	4.3		907 (1.03)	182 (1.08)	24.6 (0.31)	184.8 (1.17)	15.8 (1.16)
Wide8 CBTC	2.6 + 56.2 (1.10)	4.7		935 (1.06)	168 (1.00)	25.2 (0.32)	191.5 (1.21)	16.9 (1.24)
Wide8 Quantized	58.5 (1.10)	4.2	2.7	697 (0.79)	337 (2.01)	25.8 (0.32)	193.5 (1.22)	14.0 (1.03)
Wide8 Quantized CBTC	2.6 + 66.9 (1.30)	4.7		717 (0.81)	315 (1.87)	26.5 (0.33)	201.4 (1.27)	15.1 (1.11)
Wide8 Slang	51.9 (0.97)	4.3	3.4	721 (0.82)	168 (1.00)	24.6 (0.31)	184.6 (1.16)	15.8 (1.16)
Wide8 Slang CBTC	2.8 + 55.9 (1.10)	4.7		739 (0.84)	154 (0.92)	25.2 (0.32)	191.2 (1.21)	16.9 (1.24)
Wide8 Slang Quantized	56.7 (1.06)	4.3		578 (0.66)	288 (1.72)	25.4 (0.32)	190.2 (1.20)	16.3 (1.20)
Wide8 Slang Quantized CBTC	2.8 + 64.2 (1.25)	4.7		594 (0.68)	267 (1.59)	26.1 (0.33)	197.7 (1.25)	17.6 (1.29)

Table 16: Results of all shader variants for the san_miguel scene. Values in parentheses are relative to the Binary variant. The optimal value within each category is highlighted in bold. For the build time of view-dependent BVHs, the first value represents the elapsed time for tracing sample rays, and the second value is the total build time for both BVHs.

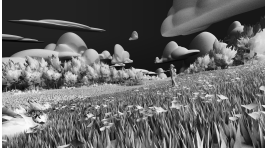
<div> <div>red_autumn_forest</div> <div>(14.44 Mtris)</div>  </div>	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)	Tested nodes per ray	Tested BVs per ray	Tested triangles per ray
Binary	68.5 (1.00)	2.0	2.4	706 (1.00)	153 (1.00)	71.1 (1.00)	142.2 (1.00)	13.2 (1.00)
Binary Slang	68.5 (1.00)	2.0		685 (0.97)	159 (1.04)	70.9 (1.00)	141.8 (1.00)	13.1 (1.00)
Wide4	69.3 (1.01)	3.1		857 (1.21)	199 (1.30)	36.5 (0.51)	142.0 (1.00)	12.9 (0.98)
Wide4 CBTC	2.0 + 80.4 (1.20)	3.2		854 (1.21)	199 (1.30)	35.5 (0.50)	138.2 (0.97)	13.2 (1.01)
Wide4 Quantized	78.0 (1.14)	3.1		804 (1.14)	327 (2.14)	37.3 (0.52)	145.1 (1.02)	13.2 (1.00)
Wide4 Quantized CBTC	2.1 + 90.3 (1.35)	3.2		814 (1.15)	329 (2.16)	36.2 (0.51)	141.1 (0.99)	13.5 (1.02)
Wide4 Slang	70.8 (1.03)	3.1		742 (1.05)	205 (1.34)	36.4 (0.51)	141.7 (1.00)	12.8 (0.98)
Wide4 Slang CBTC	2.3 + 80.9 (1.22)	3.2		745 (1.05)	205 (1.35)	35.4 (0.50)	137.9 (0.97)	13.2 (1.00)
Wide4 Slang Quantized	78.2 (1.14)	3.1		635 (0.90)	317 (2.08)	37.2 (0.52)	144.8 (1.02)	13.1 (1.00)
Wide4 Slang Quantized CBTC	2.3 + 91.4 (1.37)	3.2		642 (0.91)	320 (2.10)	36.1 (0.51)	140.8 (0.99)	13.4 (1.02)
Wide6	69.6 (1.02)	3.9	2.8	762 (1.08)	196 (1.28)	28.1 (0.39)	159.8 (1.12)	14.4 (1.10)
Wide6 CBTC	2.0 + 79.2 (1.19)	4.0		709 (1.00)	202 (1.33)	27.1 (0.38)	155.1 (1.09)	14.6 (1.11)
Wide6 Quantized	77.1 (1.13)	3.9		730 (1.03)	362 (2.37)	28.7 (0.40)	163.7 (1.15)	14.7 (1.12)
Wide6 Quantized CBTC	2.0 + 89.7 (1.34)	4.0		706 (1.00)	368 (2.41)	27.9 (0.39)	159.5 (1.12)	15.0 (1.14)
Wide6 Slang	69.5 (1.02)	3.9		689 (0.98)	204 (1.34)	28.0 (0.39)	159.4 (1.12)	14.4 (1.09)
Wide6 Slang CBTC	2.3 + 78.7 (1.18)	4.0		661 (0.94)	211 (1.38)	27.0 (0.38)	154.8 (1.09)	14.5 (1.11)
Wide6 Slang Quantized	76.4 (1.12)	3.8	3.3	564 (0.80)	307 (2.01)	28.3 (0.40)	161.0 (1.13)	16.7 (1.27)
Wide6 Slang Quantized CBTC	2.2 + 86.2 (1.29)	4.0		545 (0.77)	311 (2.03)	27.4 (0.38)	156.6 (1.10)	17.0 (1.29)
Wide8	68.4 (1.00)	4.4		673 (0.95)	173 (1.13)	24.5 (0.34)	181.8 (1.28)	16.6 (1.27)
Wide8 CBTC	2.0 + 75.7 (1.13)	4.6	2.8	678 (0.96)	172 (1.13)	24.0 (0.34)	180.0 (1.27)	17.1 (1.30)
Wide8 Quantized	77.1 (1.13)	4.4		606 (0.86)	323 (2.12)	25.5 (0.36)	189.4 (1.33)	14.9 (1.13)
Wide8 Quantized CBTC	2.0 + 90.0 (1.34)	4.6		606 (0.86)	316 (2.07)	25.2 (0.35)	189.2 (1.33)	15.4 (1.17)
Wide8 Slang	68.7 (1.00)	4.4	3.3	548 (0.78)	156 (1.02)	24.4 (0.34)	181.4 (1.28)	16.6 (1.26)
Wide8 Slang CBTC	2.2 + 75.7 (1.14)	4.6		552 (0.78)	156 (1.02)	23.9 (0.34)	179.6 (1.26)	17.0 (1.29)
Wide8 Slang Quantized	76.8 (1.12)	4.4		495 (0.70)	287 (1.88)	25.1 (0.35)	186.1 (1.31)	17.0 (1.29)
Wide8 Slang Quantized CBTC	2.2 + 86.9 (1.30)	4.6		494 (0.70)	280 (1.83)	24.7 (0.35)	185.7 (1.31)	17.5 (1.33)

Table 17: Results of all shader variants for the red_autumn_forest scene. Values in parentheses are relative to the Binary variant. The optimal value within each category is highlighted in bold. For the build time of view-dependent BVHs, the first value represents the elapsed time for tracing sample rays, and the second value is the total build time for both BVHs.