

Wide Bounding Volume Hierarchies for Ray Tracing

Lukáš Cezner*

Supervised by: Jiří Bittner†

Department of Computer Graphics and Interaction
Czech Technical University in Prague
Prague / Czech Republic

Abstract

Ray tracing, as a technique for producing realistic images, is frequently utilized for offline rendering and increasingly in real-time rendering as well. Modern ray tracing frameworks usually employ wide bounding volume hierarchies (BVH), i.e. hierarchies with branching factor more than two. This paper explores methods for constructing and traversing these acceleration structures. We implemented basic wide BVH construction and traversal using Vulkan 1.3 API and explored utilizing Slang versus GLSL for implementing the traversal shader. In the five different scenes, the evaluations demonstrated an average speedup of 20% with 4-ary versus binary BVH. Slang sometimes greatly outperforms (up to +23%) the GLSL implementation, while at other times it significantly underperforms (up to -30%).

Keywords: Ray Tracing, Wide BVH, Vulkan, Slang

1 Introduction

Realistic image generation commonly employs ray tracing, which requires casting a very large number of rays (finding their intersection with the geometry of a scene). To efficiently address this task, an acceleration structure must be used, and the bounding volume hierarchy (BVH) is commonly used.

Rendering of a scene consists of two steps: the construction of a BVH for the scene and the traversal of this BVH in order to find ray-primitive intersections. There are many types of BVHs and the methods for building them are very diverse. Notably, BVHs can be categorized into two classes: simpler binary trees and ones with a higher branching factor (node arity), commonly referred to as a wide BVH. The advantage of wide BVHs is a lower tree depth and number of nodes, resulting in lower memory consumption and potentially in performance increase.

In this paper, we evaluate an efficient method for wide BVH construction introduced by Benthin et al. [4] and implement a simple method for traversing a BVH created in this way. We compare the traversal of various types of

BVHs (binary, 4-ary, 6-ary and 8-ary BVH) implemented in GLSL, with the 4-ary BVH additionally implemented in the Slang shading language. We highlight possible pitfalls while implementing such a method and present results of measurements conducted on five testing scenes.

2 Related work

The methods and optimizations for the construction and traversal of BVH have been extensively studied. In the following paragraphs, we provide descriptions of several methods that were examined for potential implementation.

2.1 Construction

The vast majority of techniques for building a wide BVH rely on transforming an existing binary BVH, created by algorithms such as LBVH [10], TRBVH [11] or PLOC [19]. The wide BVH transformation typically represents a series of contractions by selecting an internal node, removing it from the tree, and substituting it with its child nodes. Each contraction increases the arity of a parent node, leading to the creation of a wide BVH node. An example of a node contraction is shown in Figure 1.

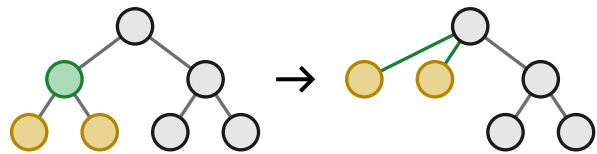


Figure 1: An example of node contraction. Upon the contraction of the green node, its child nodes (yellow) are linked directly to the parent (root node), increasing the parent's arity.

Gu et al. [8] classified tree traversal tests into *pass tests*, where a ray intersects a child's bounding volume or primitive, and *prune tests*, where no child is intersected. They noted that reducing *pass tests* speeds up traversal and introduced two heuristics: Surface-Area Guided Tree Contraction (SATC), addressing structural imbalances using the standard Surface Area Heuristic (SAH) [17], and

*lukas.cezner@email.cz (cezneluk@fel.cvut.cz)

†bittner@fel.cvut.cz

Ray-Distribution Guided Tree Contraction (RDTC), using statistical data from sample rays to account for ray-distribution imbalances [8].

Ylitie et al. [24] applied dynamic programming to optimize the internal and leaf nodes at the same time. They formulated the cost $C(N, i)$ representing the total SAH cost of the subtree at node N as a forest with up to i trees ($i \in [1; k - 1]$ for a k -ary tree). The algorithm involves two phases: a bottom-up cost computation, and then a top-down decision backtracing in which a wide BVH is created.

Benthin et al. [4] describe, as part of their HPLOC (Hierarchical Parallel Locally-Ordered Clustering) algorithm, an effective top-down wide BVH conversion method that is performed in a single kernel execution on the GPU, which is described more in detail in Section 3.

2.2 Traversal

Traversing a wide BVH is conceptually similar to a binary BVH, but requires determining the traversal order of its children. Ideally, the closest bounding volume should be traversed first, which requires sorting the intersections, e.g. using sorting networks. An example of sorting networks is shown in Figure 2.

As the arity of nodes increases, the computational complexity increases rapidly. Therefore, Ylitie et al. [24] extended the traversal order introduced by Garanzha and Loop [7], where the traversal order is defined by the XOR operation ($i \oplus r$) between a child's index i and a ray's octant r encoded as a binary number [24].

Ogaki and Derouet-Jourdan [21] introduced a technique to determine the traversal order in occlusion (any-hit) tests using ray sample statistics, sorting children in the construction phase by the likelihood that primitives in their subtree intersect a ray.

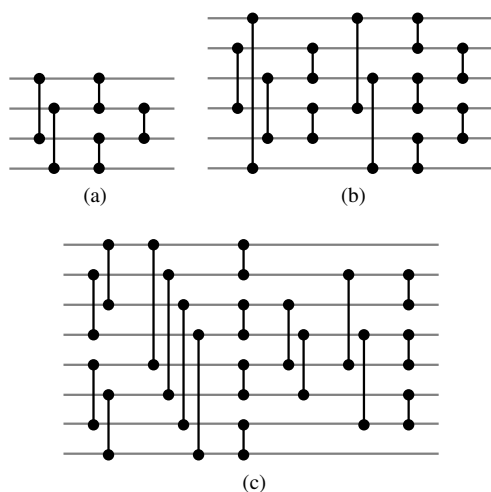


Figure 2: Optimal sorting network for various number of elements: (a) 4 elements and 5 comparisons, (b) 6 elements and 12 comparisons, (c) 8 elements and 19 comparisons [12].

3 Constructing wide BVH

For conversion from a binary BVH to a wide BVH, we chose the HPLOC single-kernel execution method [4]. This technique employs a pair of indices per shader invocation, stored in global memory: the first index denotes the binary BVH node to be processed by the invocation, while the second index points to the memory where the computed wide BVH node will be stored.

The indices are initialized to an invalid value, denoting that an invocation has not yet received a BVH node to process. Only the first invocation contains the indices of a root node. Each invocation atomically checks its index pair until it contains valid values. Subsequently, it accesses the binary BVH node at the given index and selects up to k children for the wide k -ary BVH node in the subtree, contracting nodes with the largest surface area. After that, these k children are set to be processed; one child is processed by the same invocation, while the remaining $k - 1$ are assigned to other invocations by atomically writing to not-yet-valid index pairs. This process continues until an invocation process a leaf node. Figure 3 shows this method applied to a small tree.

The selected traversal order for a wide BVH employs a sorting network, sized to match the tree's arity, as illustrated in Figure 2.

4 Implementation

The wide BVHs are implemented within the path-tracing framework named Orchard, developed by Martin Káčerik [13]. All manipulation with BVH trees, from construction to traversal, is performed on a GPU using Vulkan 1.3 compute shaders. The majority of these shaders were implemented using GLSL, with only the wide BVH traversal shader being developed both in GLSL and in Slang. Initially, a binary BVH is built using the PLOC++ algorithm [3], followed by the collapse step, which, based on the SAH cost, merges some of the leaves to increase the number of primitives per leaf. The binary BVH is later contracted into a wide BVH during the arity conversion phase via the HPLOC conversion algorithm, and lastly, the BVH is arranged for traversal during the compression step.

Path tracing utilizes a wavefront approach [14] and consists of three stages: the generation of primary rays, the BVH traversal that finds the nearest hit for all rays, and shade + cast, where intersections are resolved and secondary rays for subsequent traversal iterations are created. Between each of these stages, the GPU device synchronization is inserted. Originally, the entire path tracing utilized a single kernel execution managed by a software scheduler. However, due to the issues outlined in Section 5.1, the approach with each stage as an individual (separated) kernel execution was selected (the following text refers to this separated kernels BVH traversal, unless stated otherwise).

The BVH traversal itself is done via speculative while-while traversal with persistent threads [1], where new rays are fetched if the number of active invocations in the subgroup (32 threads) is less than 20. At present, the framework supports only first-hit traversal utilizing a diffuse reflectance model, with a maximum of eight reflections.

4.1 Shading languages

For the development of wide BVH kernels, we consider two shading languages: GLSL and Slang.

GLSL is the primary shading language for OpenGL and is frequently utilized to write shaders for Vulkan as well. Provides fundamental programming constructs such as control statements, loops, functions, and structures, but does not provide any advanced features (such as interfaces that could be used to easily represent different kinds of BVH nodes) [15]. Consequently, creating several variants of a shader requires the use of preprocessor macros, which decrease readability.

Slang shading language is a modern shading language designed to support compatibility with multiple back-ends while simultaneously providing functionalities to target a particular back-end. It is based on HLSL, commonly used in development with the DirectX API, and extends it by several modern programming features, including member functions and properties, operator overloads, interfaces, and generics, and employs a modular compilation method allowing more maintainable development of shader variants [22]. In general, Slang is a rapidly evolving and promising language, but certain features are not yet fully mature, thus it might not yet be fit for use in production environment.

4.2 Wide BVH traversal

We implement the traversal shader twice: once using GLSL and again utilizing Slang. Traversing of the wide BVH is done entirely in one shader kernel utilizing software scheduling and GPU device synchronization. In contrast, binary BVH utilizes separate kernels for traversal and ray generation for every depth (see Section 5.1).

The Slang and GLSL variants share the same BVH tree created by the GLSL construction shader. The bounding volumes of the children are stored in the parent node and intersected within the loop. Instead of loading the entire BVH node to the GPU registers at once, each bounding volume is prefetched during the prior loop iteration to mitigate memory latency without excessive use of registers, thus preventing reduced working subgroup occupancy. Subsequently, a sorting network orders the intersected BVH nodes by distance.

The Slang version of wide BVH traversal uses a traversal stack with 96 entries, compared to 64 in GLSL, as GPU drivers often store the entire stack in registers, which degrade performance due to small GPU occupancy. Table 1 presents the usage of the register, global, and shared memory for all shader variants, including those excluded from the measurement.

5 Results and Discussion

We evaluate the implementation’s performance across the five scenes: Lumberyard Bistro [2] (interior and exterior), San Miguel 2.0 [18], Red Autumn Forest [23] and Lynxs-design’s interior room [16]. Each scene was rendered from 8 different views, with 15 path samples for each pixel in

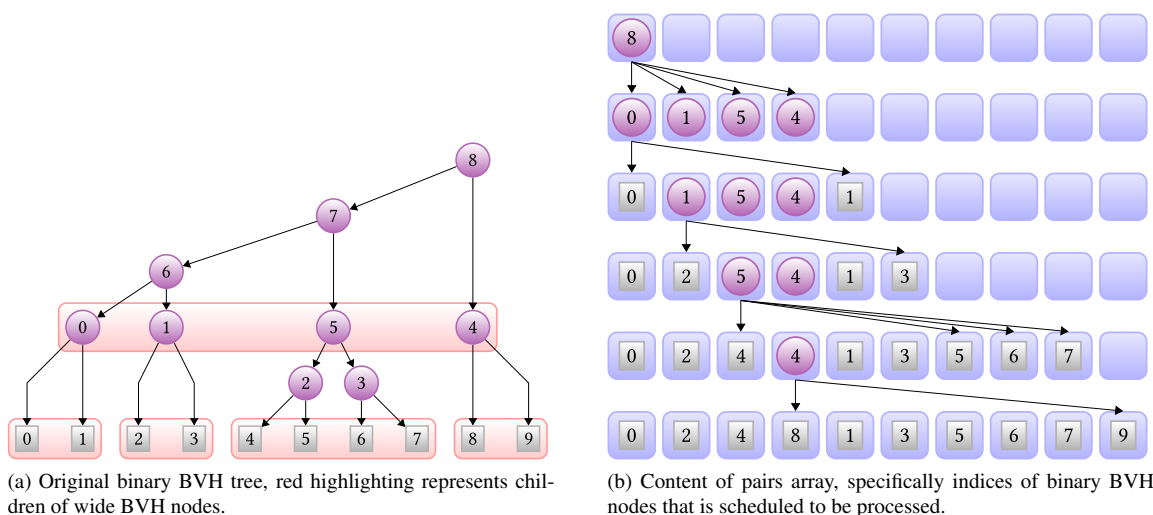


Figure 3: Example of conversion of a binary BVH tree to a 4-ary wide BVH tree with the HPLOC algorithm [4]. Starting from the binary BVH root, the first invocation determines the children for the wide BVH root. It schedules itself to process the first child (internal node 0), while the remaining children (internal nodes 1, 5, and 4) are assigned to different shader invocations. The image was taken from Benthin et al. [4]

every view. Figure 4 presents the selected view for each of these scenes. The resulting MRPs (Mega rays per second) performance is an average taken from all path samples and views within the scene.

Method	Register count	Local memory size (bytes)	Shared memory size (bytes)
Binary (single)	48	256	3344
Binary (separated)	40	256	0
Wide4 (single)	48	256	3344
Wide4 (separated)	48	256	0
Wide4 (32 × 24, sep.)	40	256	21504
Wide4 Slang (single)	156	0	4
Wide4 Slang (separated)	52	384	0
Wide6 (single)	64	256	4
Wide6 (separated)	52	256	0
Wide8 (single)	64	256	4
Wide8 (separated)	56	256	0

Table 1: Comparison of shader usage of registers, local and shared memory, gathered via Pipeline Executable Properties [6] extension. Anticipated memory consumption consists of 256 bytes (384 for the Slang variant) for the traversal stack in local memory and 4 bytes in shared memory in single-kernel mode. When register usage exceeds 40, the occupancy of compute units (SM) decreases. The workgroup size is 32×2 unless otherwise stated. The selected benchmark candidates appear in bold (they have no memory problems).

We conducted the measurements on a Linux 6.13.6 PC featuring an NVIDIA GeForce RTX 4070 Ti GPU (driver nvidia-open 570.124.04), keeping the GPU clock speed locked at 2835 MHz and the VRAM clock speed at 10501 MHz to ensure consistent results. Shaders were compiled using glslang 15.1.0 and Slang 2025.6.3. The shader execution time was measured as the difference between the start of the first invocation and the end of the last invocation, using the Shader Clock [9] extension.

Figure 5 illustrates the performance of primary and secondary rays across chosen methods. Table 5 presents the same results, along with tree construction time, scene and BVH tree statistics, in a table format. The construction of the BVH takes a similar time, where the arity conversion stage took approximately 6% of total time. An increase in the c_t parameter for 6 and 8-ary BVH results in a faster construction, as fewer BVH nodes are passed to subsequent stages (due to the larger leaf nodes).

Generally, 4-ary wide BVH performs better than other approaches (on average about 20%, and sometimes by up to 37% or even 47% in the Slang variant) for both primary and secondary rays, except in the san_miguel scene, where 6-ary BVH is superior for primary rays. In four out of five scenes, the secondary rays with 6-ary BVH yield similar results to 4-ary BVH, though it is slower for primary rays (but in different scenes). 8-ary BVHs exhibit inferior performance compared to the binary variant, likely due to the significant memory requirements (232 bytes per node) and the relatively low average of children per node, with many lower-level nodes near the leaves being only partially filled.

In three scenes, the Slang variant surpasses (up to +23%) the GLSL implementation for secondary rays, although it is poorly ranked in other cases (up to -30%), specifically in all primary rays and bistro_int’s secondary

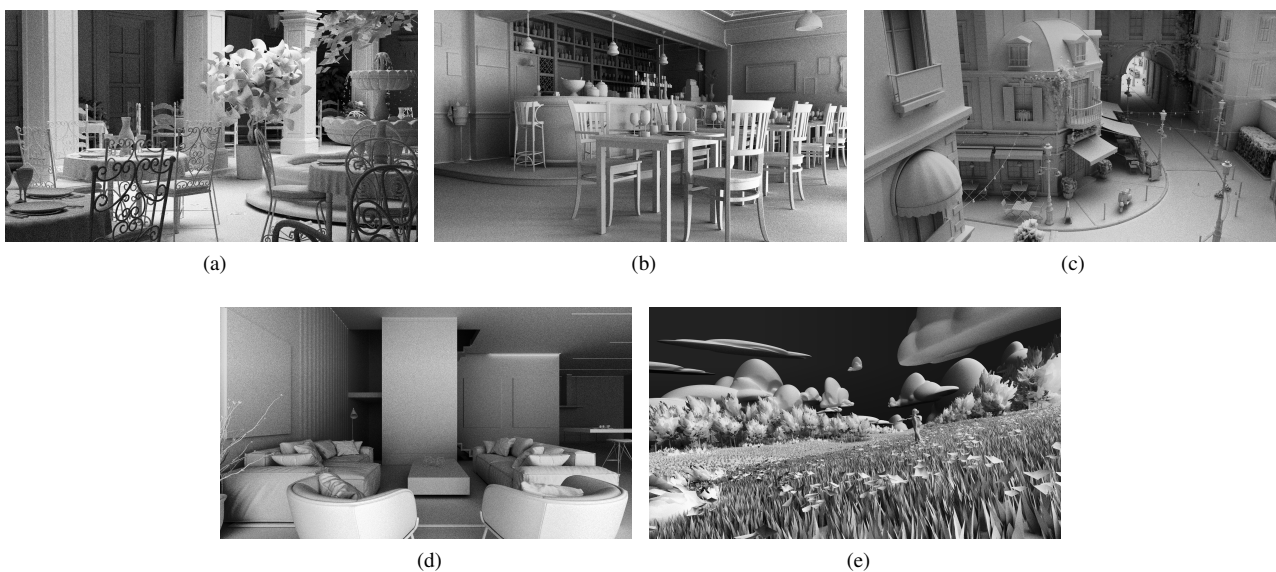


Figure 4: Rendering of data set scenes by a 4-ary BVH: (a) San Miguel 2.0 [18], (b) Lumberyard Bistro interior [2], (c) Lumberyard Bistro exterior [2], (d) Lynxsdesign’s interior room [16] and (e) Red Autumn Forest [23].

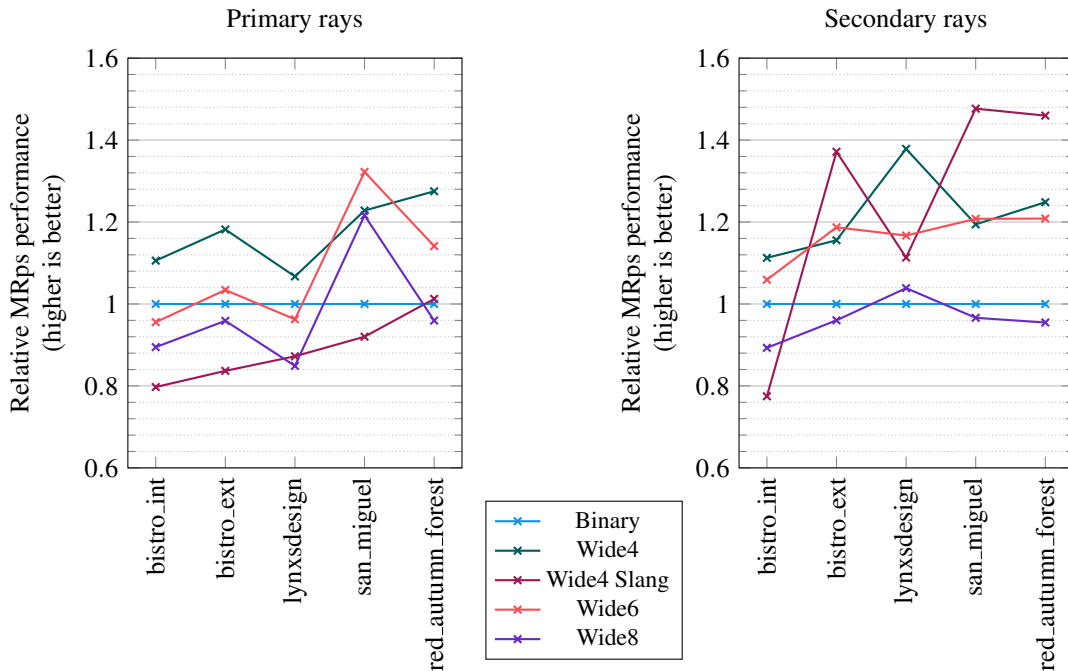


Figure 5: The relative MRps performance (the ratio of MRps in comparison to the Binary variant) of primary and secondary rays for wide BVH variants.

rays. We do not know exactly the reason for this behavior, but it can be due to Slang optimization, which moves some SPIR-V instructions before a condition [5]. Although a fix for this bug exists, it was not tested because of a limited time prior to submission of this paper.

The traversal shaders were executed using a workgroup size of 32×2 , respectively 32×32 for 6-ary BVH. The relative performance of the MRps of varying workgroup sizes is shown in Table 3. Due to significant scene variations under certain conditions, the optimal workgroup size was determined by the greatest minimum value across scenes. As shader invocations do not communicate with other threads, the impact of workgroup configuration is minor (as expected) when the GPU is adequately saturated, which usually occurs when more than two workgroups can occupy the compute unit ($< 32 \times 24$). Only 6-ary BVH benefit from the situation where only one workgroup can fit,

$c_i = 2, c_t =$	1	2	3	4	5	6
Binary	0.99	1.00	0.99	0.99	0.99	0.97
Wide4	1.00	1.00	1.00	0.99	0.97	0.96
Wide4 Slang	0.99	1.00	0.99	0.98	0.96	0.94
Wide6	0.98	1.00	1.00	1.00	0.99	0.98
Wide8	0.97	0.99	0.99	1.00	1.00	0.99

Table 2: 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.

probably due to some smaller GPU occupancy and cache accesses.

Although the change in the SAH traversal cost parameter c_t during the leaf collapse phase of the BVH construction is subtle, there is a noticeable trend: as the BVH arity increases, the optimal c_t also grows, resulting in more primitives per leaf. As indicated in Table 2, the best c_t is 2 for binary and 4-ary BVH, 3 for 6-ary BVH, and 4 for 8-ary, with c_i consistently set to 2.

5.1 Instabilities of the BVH traversal

One of the primary challenges faced in development is the instability of the single-kernel path tracer, which showed an unexpected performance fluctuation. Primarily in the Binary variant, fluctuation was in the range up to 100% – within the same execution of the program, performance varies every few seconds, which absolutely invalidates performance measurements. Our investigation suggests that this behavior probably arises from heuristic shader optimizations by the graphics driver, which seeks to reduce active registers by using temporary storage. In particular, the shader uses an extensive amount of shared memory (as much as 50 kB for a workgroup of size 32×24), despite only 4 bytes being explicitly needed. Furthermore, the high usage of shared memory limits the number of concurrent workgroups in a compute unit.

Strange profiling behavior in the NVIDIA Nsight Graphics [20] debugger further complicates troubleshooting. The first profile captured during path tracing produces

different results compared with subsequent ones. Figure 6 illustrates a comparison of profiles for the same shader during an identical session.

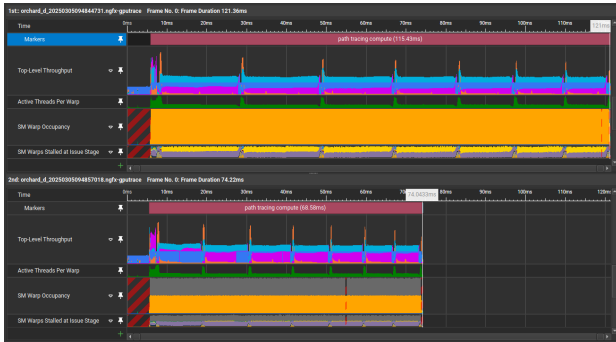


Figure 6: Comparison of two profile captures of the identical Binary single-kernel traversal shader during the same session using the NVIDIA Nsight Graphics [20] debugger. The first profile indicates full SM (compute unit) occupancy, while the second one displays occupancy limited to half the maximum, implying that the graphics driver recompiled the shader at runtime.

	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 4: The relative MRPs (average across scenes) when comparing the separated kernel against the single kernel traversal shader with the same BVH arity (values > 1 indicate that the separated kernel is faster, individual rows are independent of each other).

		32×2	32×4	32×8	32×12	32×16	32×20	32×24	32×28	32×32
Binary	min	1.00	1.00	0.99	0.98	0.97	0.95	0.96	0.96	0.98
	avg	1.00	1.00	1.00	0.99	0.98	0.97	0.97	1.01	0.99
Wide4	min	1.00	1.00	0.99	0.99	0.97	0.98	0.74	0.91	0.97
	avg	1.00	1.00	1.00	1.01	1.02	0.99	0.81	1.02	1.01
Wide4 Slang	min	1.00	0.99	0.96	0.99	0.95	0.95	0.74	0.93	0.96
	avg	1.00	1.00	0.98	1.00	0.98	1.00	0.76	0.95	0.98
Wide6	min	0.92	0.93	0.92	0.93	0.78	0.92	0.77	0.95	1.00
	avg	0.94	0.94	0.94	0.94	0.86	0.94	0.86	1.01	1.00
Wide8	min	1.00	1.00	0.98	1.00	0.97	0.83	0.86	0.93	0.97
	avg	1.00	1.00	1.00	1.00	1.00	0.89	0.98	0.99	1.00

Table 3: Relative MRPs performance with different workgroup sizes. The best workgroup size for each variant (based on the minimum value) is selected as the reference number and is shown in bold black text. Sizes where the relative MRPs are below 0.9 (which corresponds with usage of an unexpectedly large amount of shared memory) are shown in bold red text. The "min" and "avg" rows denote the minimum and average of relative MRPs across the tested scenes.

BVH traversal using separated kernels does not show these performance fluctuations. However, as indicated in Table 3 (values highlighted in red) and Table 1, certain workgroup sizes still require unusually large shared memory, resulting in a significant speed reduction.

Table 4 presents a comparison of MRPs performance between separate and single kernel variants. This comparison may not be entirely accurate due to the instabilities mentioned above, but it demonstrates a correlation with abnormal register/memory usage. Binary and both Wide4 variants exhibit a significant performance increase (which have these registry and memory issues as shown in Table 1), while Wide6 and Wide8 demonstrate identical or even reduced MRPs in the separated variants.

6 Conclusion and Future work

The conducted measurements confirm that wide BVH variants speed up raytracing compared to binary BVHs, aligning with current industry trends. More advanced methods of the wide BVH are expected to further enhance performance. Upon resolving the implementation challenges with the Slang variant of shaders, it appears that Slang can surpass commonly utilized GLSL.

Our top priority is to resolve the existing issue with slow performance of the Slang implementation in some cases. Subsequently, we should compare existing methods with more advanced techniques such as view-dependent construction heuristics and BVH tree compression.

Acknowledgments

This work was supported by the Grant Agency of the Czech Technical University in Prague, No SGS25/150/OHK3/3T/13.





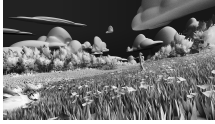
Scene	Method	Build time (ms)	Avg. children per node	Avg. primitives per leaf	Primary rays (MRps)	Secondary rays (MRps)
bistro_int (1.04 Mtris) 	Binary	13.3 (1.00)	2.0	2.1	1600 (1.00)	214 (1.00)
	Wide4	13.1 (0.98)	3.1		1770 (1.11)	238 (1.11)
	Wide4 Slang	13.0 (0.98)	3.1		1276 (0.80)	166 (0.77)
	Wide6	12.4 (0.93)	3.7	2.5	1529 (0.96)	227 (1.06)
	Wide8	12.2 (0.92)	4.3	3.3	1432 (0.89)	191 (0.89)
bistro_ext (2.83 Mtris) 	Binary	22.0 (1.00)	2.0	2.1	892 (1.00)	126 (1.00)
	Wide4	22.4 (1.02)	3.1		1054 (1.18)	145 (1.16)
	Wide4 Slang	22.4 (1.02)	3.1		746 (0.84)	172 (1.37)
	Wide6	22.0 (1.00)	3.8	2.6	922 (1.03)	149 (1.19)
	Wide8	21.0 (0.95)	4.4	3.3	855 (0.96)	121 (0.96)
lynxsdesign (8.20 Mtris) 	Binary	43.6 (1.00)	2.0	2.0	2262 (1.00)	758 (1.00)
	Wide4	47.1 (1.08)	3.0		2414 (1.07)	1045 (1.38)
	Wide4 Slang	47.0 (1.08)	3.0		1974 (0.87)	844 (1.11)
	Wide6	46.3 (1.06)	3.6	2.2	2178 (0.96)	884 (1.17)
	Wide8	44.2 (1.01)	4.1	2.7	1921 (0.85)	787 (1.04)
san_miguel (9.96 Mtris) 	Binary	58.8 (1.00)	2.0	2.1	756 (1.00)	169 (1.00)
	Wide4	62.5 (1.06)	3.1		928 (1.23)	202 (1.19)
	Wide4 Slang	62.4 (1.06)	3.1		695 (0.92)	250 (1.48)
	Wide6	59.6 (1.01)	3.7	2.7	999 (1.32)	204 (1.21)
	Wide8	58.1 (0.99)	4.3	3.4	919 (1.22)	163 (0.97)
red_autumn_forest (14.46 Mtris) 	Binary	77.4 (1.00)	2.0	2.4	675 (1.00)	165 (1.00)
	Wide4	82.3 (1.06)	3.1		861 (1.28)	206 (1.25)
	Wide4 Slang	82.2 (1.06)	3.1		683 (1.01)	241 (1.46)
	Wide6	80.2 (1.04)	3.9	2.8	770 (1.14)	199 (1.21)
	Wide8	78.3 (1.01)	4.4	3.3	648 (0.96)	157 (0.95)

Table 5: Results and statistics for benchmarked methods across five distinct scenes. The numbers in parentheses are relative values compared to the Binary variant. The best results in the category are highlighted by bold text.

References

- [1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. *Proceedings of the Conference on High Performance Graphics 2009*, August 2009.
- [2] Amazon Lumberyard. Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). <http://developer.nvidia.com/orca/amazon-lumberyard-bistro>, 7 2017. Accessed: 2025-03-05.
- [3] Carsten Benthin, Radoslaw Drabinski, Lorenzo Tesari, and Addis Dittebrandt. PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. *Proc. ACM Comput. Graph. Interact. Tech.*, 5(3), July 2022.
- [4] Carsten Benthin, Daniel Meister, Joshua Barczak, Rohan Mehalwal, John Tsakok, and Andrew Kensler. 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, August 2024.
- [5] Lukáš Cezner. ForceInline moves instructions before a condition · Issue #6654 · shader-slang/slang. [http](http://)

- [s://github.com/shader-slang/slang/issues/6654](https://github.com/shader-slang/slang/issues/6654). Accessed: 2025-04-05.
- [6] Faith Ekstrand, Ian Romanick, Kenneth Graunke, Baldur Karlsson, Jesse Hall, Jeff Bolz, Piers Daniel, Tobias Hector, Jan-Harald Fredriksen, Tom Olson, Daniel Koch, and Spencer Fricke. VK_KHR_pipeline_executable_properties(3) Manual Page. https://registry.khronos.org/vulkan/specs/latest/man/html/VK_KHR_pipeline_executable_properties.html, 5 2019. Accessed: 2025-03-31.
- [7] Kirill Garanzha and Charles Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.
- [8] Yan Gu, Yong He, and Guy E. Blelloch. Ray Specialized Contraction on Bounding Volume Hierarchies. *Computer Graphics Forum*, 34(7):309–318, 2015.
- [9] Aaron Hagan and Daniel Koch. VK_KHR_shader_clock(3) Manual Page. https://registry.khronos.org/vulkan/specs/latest/man/html/VK_KHR_shader_clock.html, 4 2019. Accessed: 2025-03-31.
- [10] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. *High Performance Graphics*, page 33–37, 6 2012.
- [11] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference, HPG '13*, page 89–99, New York, NY, USA, 2013. Association for Computing Machinery.
- [12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.
- [13] Martin Káčetik and Jiří Bittner. SAH-Optimized k-DOP Hierarchies for Ray Tracing. *Proc. ACM Comput. Graph. Interact. Tech.*, 7(3), August 2024.
- [14] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference, HPG '13*, page 137–143, New York, NY, USA, 2013. Association for Computing Machinery.
- [15] Graeme Leese, John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language, Version 4.60.8. <https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>, 8 2023. Accessed: 2025-03-05.
- [16] Lynxsdesign. Blender 4.1 Splash Screen – Lynxsdesign. <https://www.blender.org/download/demo/splash/blender-4.1-splash.blend>, 3 2024. Accessed: 2025-03-05.
- [17] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 5 1990.
- [18] Morgan McGuire. Computer Graphics Archive. <https://casual-effects.com/data>, 7 2017. Accessed: 2025-03-05.
- [19] Daniel Meister and Jiří Bittner. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1345–1353, 2018.
- [20] NVIDIA Corporation. NVIDIA Nsight Graphics. <https://developer.nvidia.com/nsight-graphics>, 2018–2024. Accessed: 2025-03-05.
- [21] Shinji Ogaki and Alexandre Derouet-Jourdan. An N-ary BVH Child Node Sorting Technique for Occlusion Tests. *Journal of Computer Graphics Techniques (JCGT)*, 5(2):22–37, June 2016.
- [22] Slang contributors. Slang User’s Guide. <https://shader-slang.org/slang/user-guide/>. Accessed: 2025-03-05.
- [23] Robin Tran. Blender 2.91 Splash Screen – Red Autumn Forest. <https://cloud.blender.org/p/gallery/5fbd186ec57d586577c57417>, 11 2020. Accessed: 2025-03-05.
- [24] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics, HPG '17*, New York, NY, USA, 2017. Association for Computing Machinery.