HIPRT: A Ray Tracing Framework in HIP

DANIEL MEISTER, Advanced Micro Devices, Inc., Japan
PARITOSH KULKARNI, Advanced Micro Devices, Inc., Canada
AARYAMAN VASISHTA, Advanced Micro Devices, Inc., Japan
TAKAHIRO HARADA, Advanced Micro Devices, Inc., USA

Fig. 1. Images rendered with applications using HIPRT (from left to right): Blender Cycles (Splash Fox), PBRT-v4 (Landscape), and Radeon ProRender (Edo).

We present HIPRT, an open-source ray tracing framework in HIP. HIPRT provides a versatile, cross-platform solution for professional rendering on contemporary many-core architectures. The core of the framework relies on the bounding volume hierarchy (BVH) with scalable construction algorithms and efficient ray traversal, employing hardware acceleration on AMD GPUs. From a user perspective, we aim at minimalist and user-friendly API design, allowing a user to write ray tracing applications only in a few lines of code. Unlike other graphics APIs that couple ray tracing and shading together, HIPRT provides only ray tracing functionality and thus can be seamlessly integrated into existing rendering environments. To demonstrate advanced features of HIPRT, we integrated it into the three rendering systems: Blender Cycles, PBRT-v4, and Radeon ProRender.

CCS Concepts: • **Computing methodologies** → **Graphics systems and interfaces**; **Ray tracing**; **Visibility**; • **Theory of computation** → **Sorting and searching**; **Massively parallel algorithms**.

Additional Key Words and Phrases: Bounding volume hierarchy, graphics systems and interface, ray tracing

ACM Reference Format:
Daniel Meister, Paritosh Kulkarni, Aaryaman Vasishta, and Takahiro Harada. 2024. HIPRT: A Ray Tracing Framework in HIP. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3 (July 2024), 18 pages. https://doi.org/10.1145/3675378

1 INTRODUCTION

With recent advances in hardware, GPU ray tracing has been gradually penetrating both professional (offline) and real-time (online) rendering. Professional rendering (e.g., the movie production) was almost exclusively dominated by CPU-based ray tracing due to limited GPU memory and costly

Authors' addresses: Daniel Meister, Advanced Micro Devices, Inc., Japan, daniel.meister@amd.com; Paritosh Kulkarni, Advanced Micro Devices, Inc., Canada, paritosh.kulkarni@amd.com; Aaryaman Vasishta, Advanced Micro Devices, Inc., Japan, aaryaman.vasishta@amd.com; Takahiro Harada, Advanced Micro Devices, Inc., USA, takahiro.harada@amd.com.

CPU-GPU transfers. However, high-end GPU models have significantly more memory at their disposal, and thus allow to render even complex scenes entirely in-core. Production renderers support various backends to maximize compute capabilities on diverse devices. There are two major GPU-based industrial ray tracing frameworks oriented on professional rendering: Embree [Wald et al. 2014] for Intel GPUs and OptiX [Parker et al. 2010] for Nvidia GPUs. We introduce HIPRT, a ray tracing framework written in the HIP[1] kernel language, tailored for professional rendering on AMD GPUs.

HIPRT is an open-source framework providing functionality for diverse scenarios commonly encountered in professional rendering environments, including multi-level instancing, motion blur, custom primitives, and intersection filters. HIPRT is entirely implemented on the GPU based on bounding volume hierarchy (BVH), accelerating ray traversal through specialized hardware units on AMD GPUs. HIPRT provides a scalable bounding volume hierarchy construction relying on optimized BVH construction algorithms. In addition, we propose a novel massively parallel construction algorithm for bounding volume hierarchy with spatial splits (SBVH), optimized to handle disproportional diagonal and oblong scene primitives. HIPRT implements the instance-based multi-segment motion blur with non-uniform time intervals. Motion blur provides a correct interpolation even for transformation matrices thanks to internal component-wise representation. HIPRT API is designed to be flexible, transparent, and lightweight. The design emphasizes providing minimal ray tracing functionality such as BVH build and ray traversal while shading and other computations are left to the application side. This design choice significantly simplifies the concepts such as *shader binding table* (SBT), making the setup of the application and data management very simple and intuitive. HIPRT is based on the HIP kernel language, supporting modern C++ standards. To sum up our contributions:

- We provide a complete ray tracing framework in a general-purpose language tailored for professional rendering on AMD GPUs.
- We propose a novel BVH construction algorithm, building high-quality BVHs with spatial splits on GPU.
- We release the source code such that it can serve as a reference either for the ray tracing research community or for engineers in the industry building their own ray tracing system.
- We also release the source codes of our PBRT-v4 port to HIP and HIPRT as an advanced tutorial, demonstrating how HIPRT can be easily integrated into an existing framework.
- We address some of the API design drawbacks of the existing APIs, such as motion blur or the shader binding table.

2 RELATED WORK

In this section, we review the most relevant work related to ray tracing APIs and bounding volume hierarchy.

2.1 Ray Tracing APIs and Renderers

OptiX [Parker et al. 2010] introduced a programmable ray tracing pipeline that was later adopted by two major graphics APIs as ray tracing extensions: DirectX [Microsoft 2020] and Vulkan [Khronos Group 2020]. While the programmable pipeline is tailored for graphics APIs, where shading and ray tracing are tightly bound, it is not well suited for professional rendering, where a renderer is typically implemented in a general-purpose language. Isolated shaders also make things opaque and difficult to debug. Last, the shader binding table (SBT) that is inherently coupled with the programming pipeline turned out to be the most challenging part to set up, giving rise to the whole

[1]Heterogeneous-Compute Interface for Portability

book chapters [Haines and Akenine-Möller 2019]. Various wrappers such as *OWL* [Wald et al. 2024] can simplify the setup of some of the features.

DirectX [Microsoft 2020] and Vulkan [Khronos Group 2020] are low-level APIs, requiring hundreds or even thousands of lines of code to render a single triangle. Both APIs are vendor-independent; however, DirectX is supported only on the Windows OS. Metal [Apple 2023] is another low-level graphics API developed by Apple, supporting hardware-accelerated ray tracing on Apple computers and mobile devices. Embree [Wald et al. 2014] has been designed from the beginning as a ray tracing framework for professional rendering, adopted by many industrial renderers. Embree, initially designed for CPUs, has been recently extended to Intel ARC GPUs that support hardware-accelerated ray tracing via SYCL.

Unlike the other two graphics APIs, Embree, OptiX, and Metal support more advanced features essential for professional rendering such as multi-level instancing, motion blur, or curve primitives. Some of these features are also available in Vulkan through vendor-specific extensions. Nonetheless, manual loading of extensions, including the ray tracing one, is an unnecessary burden on the user side. Furthermore, architectures that do not support these extensions would need a fallback, resulting in an additional code path, making the maintenance more difficult.

Separating ray tracing from rendering allows seamless switching between various ray tracing backends [Georgiev and Slusallek 2008]. Many professional renderers, including Arnold [Georgiev et al. 2018], MoonRay [DreamWorks 2023], and RenderMan [Christensen et al. 2018], implement backends using Embree or OptiX. Hence, these renderers could use HIPRT as another backend to utilize the hardware capabilities of AMD GPUs.

Mitsuba [Nimier-David et al. 2019] is a retargetable system based on Dr.JIT [Jakob et al. 2022], using CUDA and OptiX with the *parallel thread execution* (PTX) ISA to compose and compile optimized rendering kernels in runtime. Rodent [Pérard-Gayot et al. 2019] is a framework in a domain-specific language that can generate optimized rendering programs. LuisaRender [Zheng et al. 2022] encapsulates the ray tracing functionality of ray tracing engines behind a common API using an extended C++ and utilizing OptiX or DirectX as backends. Similarly to the professional renderers mentioned above, Rodent and LuisaRender could implement a HIPRT backend for AMD GPUs. Mitsuba requires an intermediate representation (e.g., PTX), which is not supported by HIP.

2.2 Bounding Volume Hierarchy

All contemporary ray tracing frameworks rely exclusively on the *bounding volume hierarchy* (BVH) as an underlying acceleration data structure, and HIPRT is not an exception. BVH has been extensively studied in the past; therefore, we limit ourselves only to the most relevant works focusing on state-of-the-art GPU-based algorithms. We refer to a survey by Meister et al. [2021] for more details.

LBVH Lauterbach et al. [2009] introduced one of the earliest construction algorithms for GPUs known as *linear* BVH (LBVH). The algorithm is based on sorting scene primitives along the Morton curve, where the order along the curve is given by Morton codes [Morton 1966]. The key observation is that the sorted Morton codes define an implicit BVH constructed by spatial median splits, where each bit defines a split. The whole BVH is constructed in an iterative manner, where in each iteration, a single level is constructed, corresponding to one kernel launch. Karras [2012] reformulated LBVH that the BVH topology can be constructed in a single kernel launch, yet still, another pass is necessary to fit bounding boxes in a bottom-up manner. This issue was addressed by Apetrei [2014], who proposed an algorithm with a single bottom-up pass that simultaneously computes topology and fits the bounding boxes. This method is considered to be the fastest algorithm to date. The standard Morton codes approximate a scene primitive by a single point that might be quite different

from its actual extent. Vinkler et al. [2017] proposed extended Morton codes that encode not only a position but also the size of a scene primitive that improves the quality of the BVH. For sorting the Morton codes, GPU variants of radix sort such as those by Merrill and Grimshaw [2011] or Adinets and Merrill [2022] can be used.

PLOC Meister and Bittner [2018] proposed *parallel locally-ordered clustering* (PLOC) constructing high-quality BVHs, iteratively merging multiple cluster pairs in parallel. Each iteration consists of three steps: nearest neighbor search, merging, and compaction. To find the nearest neighbor, the authors proposed to test clusters along the Morton curve without the necessity of any explicit data structure. In the original algorithm, each step is implemented as one kernel launch. Benthin et al. [2022] proposed to fuse all three steps into a single kernel launch with a couple of other optimizations, achieving construction performance close to LBVH while maintaining the quality of the original PLOC algorithm.

SBVH Spatial splits may achieve tighter boxes at the cost of increasing the number of references to scene primitives. This can improve the BVH quality significantly, especially for scenes with overlapping, oblong, or diagonal scene primitives. Ernst and Woop [2011], Dammertz and Keller [2008], and Karras and Aila [2013] propose to pre-split triangles before the actual construction, which is fast and easy to parallelize, but the quality improvement is rather marginal as each triangle is processed individually not taking into account neighboring triangles. Stich et al. [2009] and Popov et al. [2009] proposed to allow spatial splits during the top-down construction, selecting either (standard) object or spatial split based on the local BVH cost approximation [Meister and Bittner 2022]. Fuetterling et al. [2016] proposed a scheduling strategy based on dynamic thread pools to construct SBVH in parallel on the CPU. As a top-down algorithm, this approach puts stress on top levels that are more important than lower ones as they are visited during the traversal by most of the rays. However, top-down construction is difficult to adapt for GPU, and thus, it is considered slow.

Wide BVH All modern GPU ray tracing engines adapted the concept of *wide BVH* [Guthe 2014; Lier et al. 2018; Ylitie et al. 2017] (i.e., a BVH with a higher branching factor). The ray-triangle and ray-box intersections can be computed in parallel over multiple SIMD lanes, but what is more important for the GPU is that the wide BVH reduces the number of visited nodes during ray traversal, and thus, reducing the memory traffic. In practice, we build a binary BVH (all the aforementioned methods produce binary BVHs), and then we convert it to a wide BVH by pulling child nodes to the parent nodes in a top-down manner [Wald et al. 2014]. Pinto [2010] and Ylitie et al. [2017] proposed an algorithm based on dynamic programming that performs this conversion optimally with regard to the BVH cost. The direct construction of wide BVHs is considered difficult, remaining as an open problem.

Two-level hierarchy Wald et al. [2003] introduced the concept of a two-level hierarchy, where a bottom-level BVH is built for each object in the scene, and then a single top-level BVH is built over all objects. This concept supports the instancing of scene objects by referencing a bottom-level BVH with an affine transformation from the leaves of the top-level BVH. This concept is also useful for dynamic geometry. If an instantiated object (i.e., the transformation changes) moves, we need to update only the top-level BVH, which typically comprises only a fraction of the whole hierarchy. Nonetheless, the two-level hierarchy decreases the performance if instances significantly overlap. Benthin et al. [2017] proposed *rebraiding* (i.e., opening nodes of the bottom-level BVHs and pulling them up to the top-level BVH), decreasing the overlap and improving the overall BVH quality. To construct the top-level BVH, we need to know the bounding boxes of the transformed instances. The straightforward approach is to compute axis-aligned bounding boxes of the transformed corners of

the root bounding box of the instantiated bottom-level BVH. However, the resulting bounds might be too conservative. Laine and Karras [2015] proposed the apex point map method that allows to compute tighter bounds for instances.

Ray Traversal Aila and Laine [2009] proposed a stack-based ray traversal with persistent threads and dynamic fetch for binary BVHs. Guthe [2014] extended this algorithm for 4-wide BVHs, showing that a higher branching factor helps hide latency. Later, Ylitie et al. [2017] showed that a compressed 8-wide BVH can reduce memory traffic even further, improving the overall ray tracing performance. Lier et al. [2018] experimented with SIMD ray traversal for wide BVHs on GPUs.

In the context of GPU ray traversal, various stackless algorithms have been proposed. Laine [2010] and Vaidyanathan et al. [2019] proposed a restart trail technique for binary BVH and wide BVHs, respectively, skipping already processed subtrees while traversing every time from the root. Another class of stackless algorithms is based on backtracking with parent links. Hapala et al. [2013] proposed a stackless algorithm based on simple state logic. A caveat is that the state logic needs to evaluate the order of child nodes multiple times, which limits its practicality to simple heuristics such as those based on a sign of the ray direction. Barringer and Akenine-Möller [2013] and Áfra and Szirmay-Kalos [2014] proposed to encode the order into a bit trail for binary BVH and wide BVHs, respectively, without the necessity to evaluate the child order multiple times. Binder and Keller [2016] proposed backtracking in constant time based on a path to the node in a bitset and perfect hashing.

3 HIPRT API OVERVIEW

Similarly to Embree [Wald et al. 2014], we are not restricted by standards defined by the third parties. While designing HIPRT API, we keep two goals in mind. First, the API needs to be intuitive and easy to integrate into existing rendering frameworks. Second, the API must be general purpose, covering all possible functionality required by modern professional renderers.

```
// Triangle mesh
hiprtTriangleMeshPrimitive mesh;
mesh.triangleIndices = ...;
mesh.vertices = ...;
...

// Build input
hiprtGeometryBuildInput input;
input.type = hiprtPrimitiveTypeTriangleMesh;
input.triangleMesh.primitive = mesh;

// Create and build geometry
hiprtGeometry geom;
hiprtCreateGeometry(..., input, ..., geom);
hiprtBuildGeometry(..., input, ..., geom);

// Build trace kernel
hipFunction_t func;
hiprtBuildTraceKernels(..., &func, ...);
```

Listing 1. An example of using the HIPRT API on the host side, building a single geometry and a trace kernel. First, we create a triangle meshed defined by vertices and indices. Then, we assign the mesh to the build input, create (allocate) and build geometry. Lastly, we build a trace kernel, which is returned as a standard HIP function.

Initializing HIPRT is as simple as a single line where we create the HIPRT context by passing the HIP context and the HIP device. HIPRT defines two acceleration structure types: *geometry* and *scene*. Geometry is the bottom-level acceleration structure (BLAS) built either over triangles or

custom primitives. The triangles are defined as a triangle mesh with vertices and indices; the custom primitives are defined as a list of axis-aligned bounding boxes. Scene is the top-level acceleration structure (TLAS) built over instantiated geometries or other scenes in the case of more than two levels (see Section 4.1). Each instance has one or more affine transformations in the case of per-instance motion blur. The transformation can be specified either as a 3×4 matrix or as an SRT frame (i.e., scale, rotation, and translation). For both types, we can specify the time, which allows us to define non-uniform intervals for motion blur.

Single or multiple geometries or scenes can be created, built, compacted, and updated via the host HIPRT API (see Listing 1). A user can specify the preferred build algorithm: fast, balanced, or high-quality. This allows us to balance construction speed and ray tracing performance for a particular use case. One interesting feature that HIPRT supports is importing a custom BVH. The build input structure has an optional list of nodes defining a topology and bounding boxes of the custom BVH. If the list is specified, then HIPRT skips the build and just converts the nodes from the API format to the internal format. This feature might be useful for research, allowing users to benchmark their BVH builders with HIPRT hardware-accelerated kernels.

HIPRT supports two types of custom functions: an intersection function for custom primitives and a filter function for filtering intersections that are useful, for example, for alpha masking or filtering self-intersections. The custom functions are organized into a 2D table (ray types $\times$ geometry types). The geometry type is a user-defined integer that can be specified in the geometry build input. The ray type is specified during the ray traversal. The trace kernel is a standard HIP device kernel. However, it must be compiled via the HIPRT API to construct the custom function table and inject the necessary ray tracing functionality to the user code (see Listing 1). The custom functions must be device functions in the same module as the trace kernel.

A geometry or scene can be passed as a kernel argument. In the kernel itself, we create a HIPRT traversal object (either for the closest hit or for any hit), passing a ray to be traced and geometry or scene together with other arguments (e.g., the ray type, ray mask, etc.). We can also pass a traversal stack type as a template argument. This allows us to have various traversal stack implementations (see Section 4.2) tailored for diverse scenarios.

```
__global__ void RayTraceKernel(hiprtGeometry geom, ...)
{
    // Generate ray
    hiprtRay ray = generateRay(...);

    // Traversal object
    hiprtGeomTraversalClosest tr(geom, ray, ...);

    // Find hit
    hiprtHit hit = tr.getNextHit();

    // Do whatever you want with the hit
    ...
}
```

Listing 2. An example of using the HIPRT API on the device side. We create a ray and the traversal object that we use to find the closest intersection.

We find a hit by calling the traversal object (see Listing 2). There are two types of traversal objects to find the closest hit and any hit. The latter one can be used for visibility queries or to find multiple hits by calling the traversal object repeatedly. It is up to the application how the found intersections are used. The hit structure contains primitive ID, instance ID(s), ray distance, barycentric coordinates, and normal in the object space. In the case of multi-level instancing, there are multiple instance IDs (one per level); the first component corresponds to the top-most scene, the

component corresponds to the next level, and so on. As the scene contains transformations across all levels and all time steps, HIPRT API provides a set of functions to query such transformations based on instance IDs and at a given time. This is very convenient as users do not need to compose and interpolate transformation manually by themselves.

4 HIPRT IMPLEMENTATION

In this section, we describe the implementation details of HIPRT, focusing on the BVH construction and ray traversal.

4.1 BVH Construction

At the core of HIPRT, there are three BVH builders corresponding to the three quality levels (see Section 3): LBVH (fast), PLOC (balanced), and SBVH (high-quality). The are two steps common for all builders: triangle pairing (for triangles only) at the beginning and conversion to a wide BVH at the end.

Triangle Pairing The BVH format allows us to store triangle pairs in leaf nodes [AMD 2023b]. Thus, we merge triangles into pairs in an optional preprocessing step. This can be efficiently implemented through the warp-level primitives [Meister et al. 2023], restricting the search to a single warp, assuming that the order of triangles is not completely random and that the neighboring triangles are stored closely to each other. This works surprisingly well, decreasing the input for the actual build by about 30% on average and thus significantly accelerating all further steps.

LBVH We use 32-bit Morton codes that adaptively adjust the number of bits for indiviual dimensions based on the scene extent [AMD 2023; Vinkler et al. 2017]. For sorting, we use radix sort from the Orochi library [AMD 2022], based on the one-sweep algorithm [Adinets and Merrill 2022]. The build itself is based on the algorithm by Apetrei [2014], constructing the BVH in a single kernel launch. The algorithm proceeds from leaves to the root while keeping the ranges of the Morton codes of the corresponding subtrees. The parent index is either one position before the first Morton code or one position after the last Morton code in the range.

PLOC As for LBVH, we need to compute and sort Morton codes that are used to find the nearest neighbors for clustering. Our algorithm is based on PLOC++ [Benthin et al. 2022], implementing all proposed algorithmic optimizations. The most important optimization is to fetch additional clusters corresponding to the previous or following block to be able to verify the merging condition (i.e., the nearest neighbors of both clusters mutually correspond) without the necessity of communications with other blocks. This allows us to process each block completely independently, and thus, all three steps of the original algorithm can be implemented in a single kernel. Note that we still need a global synchronization after each iteration. Another important optimization is to use a unidirectional search with one atomic operation (in the shared memory) to propagate the search results to the neighbors. The last notable optimization is switching to a specialized kernel that efficiently builds the remaining top levels.

SBVH The SBVH algorithm by Stich et al. [2009] improves the BVH quality significantly, but the construction is slow as this algorithm is sequential and designed for CPUs. We adapt the SBVH algorithm for GPUs to accelerate the construction speed. Our priority is to preserve the quality of the original algorithm. Our algorithm builds the BVH iteratively, level by level, in a top-down fashion. Similarly to Garanzha et al. [2011], each iteration consists of multiple kernel launches, where either tasks (e.g., node splitting) or references (e.g., binning) are processed in parallel, starting with a single task corresponding to the root node.

Each task has two sets of bins (i.e., for object splits and spatial splits); we specify the minimum and maximum numbers of bins per task (we use 8 and 32, respectively, for each dimension). We allocate only the minimum number of bins for each potential task. The actual number of bins is scaled by the number of tasks before each iteration such that we can use more bins if there are fewer tasks and vice versa. We specify the maximum number of additional references to scene primitives (we use 50%) to allocate all necessary buffers only once; each reference has an index to the corresponding node. The number of potential tasks is bounded by half of the number of references, which corresponds to the largest possible cut in the tree, excluding the leaves. For each such potential task, we allocate two sets of bins. The tasks are stacked in a single buffer, where the task index coincides with the node index. We keep an offset and task count for each iteration. We use double buffering for active reference indices, swapping the input and output after each iteration, while references themselves are stacked in a single separate buffer.

At the beginning of each iteration, we reset both sets of bins. We perform the object binning, where references are projected to the appropriate bin of the corresponding node in parallel. We evaluate the best object split, and based on the overlap of child bounding boxes, we decide whether to perform spatial binning as in the original algorithm [Stich et al. 2009]. If the surface area of an intersection of child bounding boxes is larger than a fraction of the surface area of the root bounding box , corresponding to the α parameter in the original algorithm [Stich et al. 2009] (we use $\alpha = 10^{-4}$), then we perform spatial binning. We evaluate the best spatial splits, and we select either the best object or spatial split, choosing the one that minimizes the *surface area heuristic* (SAH). We write the node and create child tasks for the next iteration. In the last step, we split the references and redistributed them to new tasks. If the maximum number of references is to be exceeded, we turn spatial splits off. If all references fall into the same bin, we split them into two halves of roughly the same size (not necessarily sorted). For triangles, we perform the split taking into account the actual triangle geometry, while for instances and custom primitives, we split only the bounding boxes [Hendrich et al. 2017]. Note that we do not split triangle geometry but cover the triangle with multiple tighter bounding boxes.

Our algorithm implements all the same steps as the original algorithm [Stich et al. 2009] except *unsplitting*, which would make the GPU implementation more complicated. The major bottleneck is spatial binning, which is implemented via atomic updates. For standard object binning, if one increases the number of bins, it helps to distribute atomic pressure across different locations. However, this is not the case of spatial binning, where increasing the number of bins makes it worse. For example, if a triangle spans across the whole extent, it is diced into all bins. We use a small optimization; if all references in the block belong to the same task, we aggregate binning in the shared memory and then update the corresponding bins in the global memory only once. We also experimented with the build-from-hierarchy to be able to perform binning always in the shared memory, but this always leads to significant quality degradation. The issue is that if we bin triangles in clusters, we cannot perform splitting directly on the triangle geometry.

Conversion to Wide BVH The builders produce binary BVH in a compact format with 32B internal nodes. The last step is to convert the BVH to the 4-wide BVH that can be fed to ray tracing hardware units on AMD GPUs. The 4-wide BVH node contains child indices and child boxes (128B) [AMD 2023b]. This conversion is a relatively difficult problem, often omitted in research papers (e.g., shortcomings such as that it is done on CPU). The conversion based on dynamic programming [Ylitie et al. 2017] provides an optimal result, but it requires an additional bottom-up pass to compute the partial costs that have to be stored in an additional buffer. We opt for a simple solution based on surface areas using a single top-down pass. In each internal node, we replace a child node with the highest surface area by its children; we repeat this until all slots are occupied

Fig. 2. The Moana Island scene [Disney 2016] rendered in the PBRT-v4 with HIPRT on AMD Radeon PRO W7900. The scene contains 156 million unique primitives and 31 billion instantiated primitives. The scene is organized into a three-level hierarchy.

(or we reach leaves). This can be implemented on GPU using one kernel launch per level. As an optimization, we use a specialized kernel, building the first top levels by a single thread block in a single kernel launch.

Compaction We need to allocate the node buffer beforehand. Thus, we need to conservatively estimate the number of 4-wide nodes. This can be bounded as $\lceil \frac{2n-1}{3} \rceil$, where n is the number of leaf nodes. This corresponds to the worst case for the conversion above, where each internal node in the level above leaves contains only two leaf nodes. The best case is $\lceil \frac{n-1}{3} \rceil$ for 4-wide BVHs (or $\lceil \frac{n-1}{k-1} \rceil$ for k-wide BVHs in general), which corresponds to the case with no empty slots. However, even this estimate might be too conservative. Therefore, the BVH buffer can be compacted, removing the unused space. It is beneficial to work with the compact binary format before conversion as it reduces memory traffic, and the buffer with 4-wide nodes can be used as an auxiliary buffer for intermediate computations during the construction.

Multi-Level Instancing Geometries and scenes can be instantiated in multiple levels, which reduces the memory requirements significantly. For example, we are able to render the Moana Island scene with multi-level instancing in-core on a single GPU (see Figure 2). The organization into multiple levels also has logical meaning; for example, triangles are organized into meshes sharing the same materials, the meshes are further organized into objects, and objects are organized into a scene. To compute the bounds of the instances, we open the geometry and transform the bounding boxes of the grandchildren from which we compute an axis-aligned bounding box. This provides much tighter bounds than transforming the root bounding box, and it is independent of the underlying geometry size. Note that we do not use explicit rebraiding [Benthin et al. 2017]; however, spatial splits in SBVH performed on instances achieve similar results. The transformations are converted to an internal component-wise format using the QR decomposition. This allows us to correctly interpolate transformations for motion blur (see Figure 3).

Batch Construction Each geometry or scene can be rebuilt (using the same storage) or updated by refitting bounding boxes. For small geometries and scenes (up to 512 primitives or instances), we have a specialized construction, building multiple small geometries and scenes, respectively, in a single kernel launch. Internally, each geometry or scene is mapped to a single thread block. This is useful for extreme cases such as the one depicted in Figure 4, where each hair strand is represented as a single geometry.

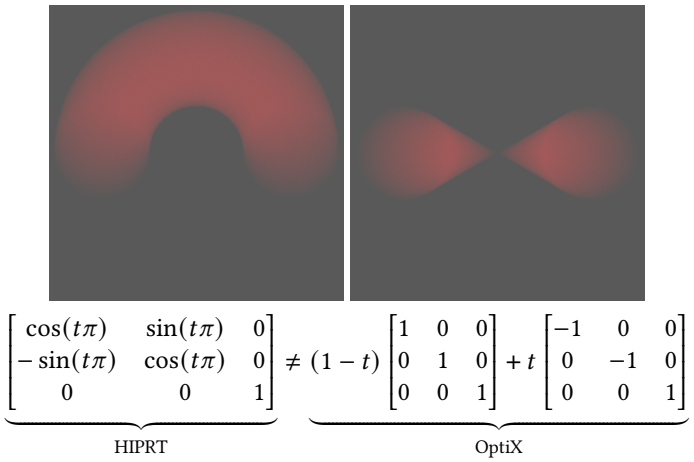

Fig. 3. HIPRT (left) correctly interpolates rotations even with transformation matrices thanks to the internal component-wise representation, unlike OptiX (right), which directly interpolates transformation matrices. Notice that OptiX provides a singular matrix for $t = 0.5$.

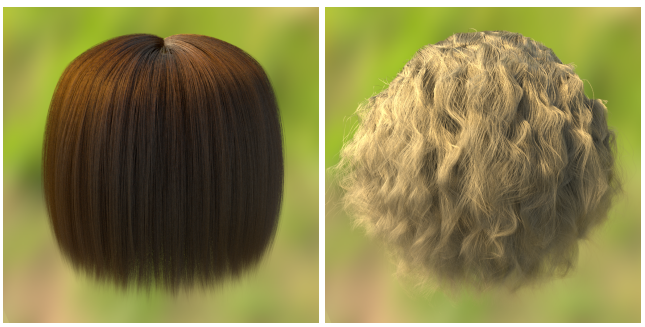

Fig. 4. Hair models rendered in the PBRT-v4 with HIPRT on AMD Radeon PRO W7900, consisting of 1.1 million (left) and 3.3 million (right) hair strands, where each hair strand is diced into 25 segments, represented as a single bottom-level geometry.

4.2 Ray Traversal

The ray traversal techniques discussed in Section 2.2 almost exclusively use a one-level hierarchy to achieve the highest possible performance. For example, *while-while* traversal by Aila and Laine [2009] is not applicable in the case of a two-level hierarchy as instead of two cases since we now have three conditions in the case of two levels or more. A general-purpose ray tracing engine must also support all possible features required by various rendering applications, which naturally brings some overhead. Thus, the main loop of ray traversal in HIPRT consists of three cases based on the current node type: internal node, leaf node (in geometry), or instance (in scene). On AMD RDNA 2 and RDNA 3 GPUs, we employ hardware intrinsics for ray-box and ray-triangle intersections [AMD 2023b]; for other architectures, we use software equivalents of the hardware intrinsics. Besides the intersection intrinsics, all other parts are implemented in software, which provides us flexibility in designing the traversal algorithm. On the other hand, to achieve high-performance, we need to consider special solutions. As each ray is traced separately, we also cannot make any assumptions

about other rays processed by threads from other warps. For instance, we cannot use a block-wise barrier inside the traversal code, as we do not know which threads are active (we still can use warp-level primitives [Meister et al. 2023]).

Custom Traversal Stack One unique feature of HIPRT is that ray traversal works with a generic traversal stack defined by an interface; the stack type is a template argument of the traversal object. This allows users to select a traversal stack based on a particular use case. HIPRT provides three types of traversal stacks: *private stack*, *global stack*, or *dynamic stack*. The private stack is the simplest type using the (private) local memory as a storage (similarly to Aila and Laine [2009]). The default ray traversal objects internally use the private stack (see Listing 2). Local memory may cause unnecessary register pressure. The global stack is the most efficient type, combining shared memory with global memory as a fallback. The shared memory buffer keeps the top-most entries (i.e., a circular buffer), while the global memory serves as a backup for the bottom-most entries. A caveat is that the global buffer must be allocated for all scheduled threads, which must be wasteful as only a fraction runs simultaneously on the GPU. This could be bypassed by persistent threads, but in complex rendering frameworks, it might be difficult to change scheduling. The dynamic stack combines shared and global memory in the same way as the global stack. What is different is how the global buffer is allocated. In the case of dynamic stack, we allocated only a limited number of stacks (not for all scheduled threads), and then we dynamically distributed them to the active threads Meister et al. [2023]. The dynamic stack introduces an additional overhead compared to the global stack, but it still might be useful on memory-critical systems.

Multi-Level Hierarchy For multi-level hierarchies, we need additional logic to keep information to be able to backtrack to upper levels. Specifically, we need to store a ray and a pointer to the scene above for each additional level. The ray could be theoretically transformed inversely, but it would likely lead to numerical errors, accumulating the error through multiple levels. We use an additional short stack to store this extra information in the case of more than two levels.

Custom Function Table HIPRT supports custom intersections and intersection filters. The custom intersection is invoked whenever a custom primitive is encountered; the intersection filter is invoked whenever an intersection is found. A function that dispatches custom functions based on the geometry type and ray type is generated during the compilation and then linked together with the traversal code to the user code. We experimented with general function pointers, which turned out to be highly inefficient (causing high register pressure). Thus, we opt for the dynamic compilation that exposes the whole function to the compiler to optimize the code properly. The custom function table structure contains a user-specified buffer specified at the trace kernel compilation type (see Section 3), typically encapsulating the geometry of custom primitives. During the traversal, based on the geometry and ray types, the appropriate table entry is selected. Similar to OptiX, a user-defined *payload* structure can be passed for additional outputs.

4.3 Technical Details

The HIPRT codebase is relatively small, comprising about 17 thousand lines of code, thanks to modern C++ features such as templates and compile-time conditions. This is especially useful for BVH construction and ray traversal, where we have different variants with only minor changes (e.g., geometry vs. scene). We want to emphasize the importance of compile time conditions for GPUs; this allows us to instantiate separate optimized variants (in the spirit of RTfact [Georgiev and Slusallek 2008]), unlike runtime conditions and virtual functions calls that often introduce additional register usage.

Given that HIPRT host code can be compiled for both AMD and NVIDIA platforms, if a user wants to compile their code for both platforms, they would have to compile twice. To alleviate this issue, HIPRT internally uses the Orochi library AMD [2022] that allows compiling the host side into a single binary which enables runtime switching on the host side between AMD and Nvidia platforms. Note that an application using HIPRT may use either Orochi or standard HIP. HIPRT can be compiled into any GPU that supports HIP, including AMD MI GPUs (unlike ray tracing APIs in Vulkan or DirectX). Note that MI GPUs do not support hardware-accelerated ray tracing. The trace kernels can be either compiled and linked via the HIPRT API at runtime or offline using HIPCC from the command line. In the case of offline compilation, we need to implement the dispatch function manually (see Section 4.2). The offline compilation is useful for large frameworks where we need to include C++ standard header files (which is not possible with runtime compilation).

HIP does not support an intermediate representation (IR) such as SPIR-V or PTX, which makes shipping rather cumbersome as we have to compile each architecture separately. The device binaries compiled by different versions of HIP SDK (also known as ROCm) are generally incompatible. For instance, the HIPRT bitcode compiled with an older version cannot be linked to an application code compiled by a newer version. To overcome these issues and make HIPRT more accessible, we have decided to open-source it.

5 EVALUATION

We evaluated HIPRT in our in-house HIP renderer. We use a wavefront path tracer that isolates trace calls from shading into separate kernel launches. This allows us to implement various ray tracing back ends independent of the shading. All experiments were conducted on AMD Radeon PRO W7900 with 48 GB VRAM with ROCm 5.7 and Vulkan 1.3.261.

Vulkan is the only cross-platform solution for AMD GPUs, and thus, we use it as a main reference method. We test both available performance options: Vulkan Fast (fast build) and Vulkan HQ (fast trace). We also employ the custom BVH import (discussed in Section 3), importing BVHs constructed by Embree with the high-quality option and spatial splits (on the CPU).

The main results are summarized in Table 1. We use ten scenes of various complexity both in geometry and light transport. We report build times, trace times for different ray types, and the SAH cost. For each scene, since the scene does not contain instances, we collapse all scene objects into a single geometry by pre-transforming the objects into a common coordinate system to report a single build time and SAH cost.

Build Times LBVH is the fastest method overall, $1.4 - 3.4\times$ faster than Vulkan Fast. PLOC is $1.3 - 4.1\times$ faster than Vulkan HQ and faster than Vulkan Fast for large scenes (with more than 2.5M triangles). Note that the most costly phase of LBVH and PLOC is the conversion to 4-wide BVH, taking more than 50% and 33% of the total build times for LBVH and PLOC, respectively. SBVH is the slowest method overall, $3.7 - 8.9\times$ slower than Vulkan HQ. However, this cost can be easily amortized by tracing more samples per pixel (see Figure 6).

Trace Times The trace times are average values per one sample per pixel. We report trace times not only for the pre-transformed single geometry but also trace times with the original two-level scene partitioning. While for the pre-transformed case, Vulkan and HIPRT are on par, we can observe that for some scenes, Vulkan is significantly slower in the original two-level partitioning, especially for shadow and secondary rays. For instance, PLOC is $1.7\times$ faster for secondary rays in the Museum scene and $1.8\times$ faster for shadow rays in the Yokohama scene. SBVH and Embree consistently achieve the lowest trace times. One outlier is the Opera House scene, where LBVH in the pre-transformed case, despite the relatively low SAH cost, is significantly slower than other methods, not reflecting the actual number of tested nodes (see Table 2). This is due to insufficient

Table 1. Performance comparison of HIPRT and the reference methods: Vulkan Fast (fast build), Vulkan HQ (fast trace), and Embree (high-quality). The values in the parentheses are relative values with the respect to HIPRT PLOC. The trace times are average trace times per sample per pixel. The SAH cost is a sum of the surface areas of leaves and internal node divided by the surface area of the root.

	Trains 	Bistro Int. 	Hangar Ship 	Opera House 	Bistro Ext. 	Museum 	Sci-fi 	Zero Day 	Toy Shop 	Yokohama 	
#triangles	836k	1207k	1235k	2512k	2829k	3650k	4809k	5165k	5212k	8217k	avg. rel. val.
Build time (pre-transformed) [ms]											
HIPRT LBVH	**2.39 (0.37)**	**2.88 (0.4)**	**4.08 (0.44)**	**4.62 (0.52)**	**5.16 (0.43)**	**6.74 (0.5)**	**8.31 (0.52)**	**12.79 (0.56)**	**8.47 (0.5)**	**12.77 (0.56)**	**0.48**
HIPRT PLOC	6.4 (1.0)	7.26 (1.0)	9.32 (1.0)	8.93 (1.0)	11.93 (1.0)	13.54 (1.0)	16.01 (1.0)	22.98 (1.0)	16.82 (1.0)	22.98 (1.0)	1.0
HIPRT SBVH	68.06 (10.63)	87.83 (12.1)	118.65 (12.73)	96.34 (10.79)	188.38 (15.79)	210.74 (15.56)	320.08 (19.99)	422.3 (18.38)	281.68 (16.75)	354.66 (15.43)	14.81
Vulkan Fast	3.44 (0.54)	5.51 (0.76)	5.68 (0.61)	12.23 (1.37)	14.09 (1.18)	18.56 (1.37)	25.86 (1.62)	27.87 (1.21)	27.18 (1.62)	44.25 (1.93)	1.22
Vulkan HQ	8.06 (1.26)	12.37 (1.7)	13.4 (1.44)	26.09 (2.92)	32.05 (2.69)	41.11 (3.04)	54.85 (3.43)	62.23 (2.71)	61.32 (3.65)	95.32 (4.15)	2.7
Primary ray trace time (pre-transformed) [ms]											
HIPRT LBVH	1.46 (1.35)	1.56 (0.93)	1.42 (1.12)	159.13 (47.79)	2.22 (1.14)	1.42 (1.29)	1.51 (1.18)	1.3 (1.2)	1.86 (1.49)	1.73 (1.15)	5.86
HIPRT PLOC	1.08 (1.0)	1.68 (1.0)	1.27 (1.0)	3.33 (1.0)	1.94 (1.0)	1.1 (1.0)	1.28 (1.0)	1.08 (1.0)	1.25 (1.0)	1.5 (1.0)	1.0
HIPRT SBVH	0.93 (0.86)	**0.9 (0.54)**	**0.87 (0.69)**	**1.91 (0.57)**	**1.28 (0.66)**	**0.89 (0.81)**	**0.99 (0.77)**	0.99 (0.92)	1.49 (1.19)	0.98 (0.65)	**0.77**
HIPRT Embree	**0.91 (0.84)**	0.91 (0.54)	1.24 (0.98)	1.95 (0.59)	1.61 (0.83)	0.91 (0.83)	**0.99 (0.77)**	**0.94 (0.87)**	**1.09 (0.87)**	**0.94 (0.63)**	**0.77**
Vulkan Fast	1.63 (1.51)	1.96 (1.17)	1.54 (1.21)	9.35 (2.81)	2.13 (1.1)	1.54 (1.4)	1.76 (1.38)	1.59 (1.47)	1.53 (1.22)	1.72 (1.15)	1.44
Vulkan HQ	1.32 (1.22)	1.8 (1.07)	1.45 (1.14)	2.23 (0.67)	1.81 (0.93)	1.2 (1.09)	1.41 (1.1)	1.13 (1.05)	1.37 (1.1)	1.44 (0.96)	1.03
Primary ray trace time (two-levels) [ms]											
HIPRT LBVH	1.74 (0.91)	2.55 (1.06)	3.51 (1.04)	4.41 (1.37)	3.57 (1.13)	1.42 (1.18)	1.63 (1.12)	1.6 (1.23)	2.53 (1.24)	1.96 (1.15)	1.14
HIPRT PLOC	1.91 (1.0)	2.41 (1.0)	3.38 (1.0)	3.21 (1.0)	3.15 (1.0)	1.2 (1.0)	1.46 (1.0)	1.3 (1.0)	**2.04 (1.0)**	1.71 (1.0)	1.0
HIPRT SBVH	**1.26 (0.66)**	**2.09 (0.87)**	**2.94 (0.87)**	2.31 (0.72)	3.15 (1.0)	1.12 (0.93)	**1.31 (0.9)**	1.58 (1.22)	2.74 (1.34)	1.63 (0.95)	0.95
HIPRT Embree	1.56 (0.82)	2.75 (1.14)	3.12 (0.92)	**2.27 (0.71)**	3.23 (1.03)	**1.1 (0.92)**	1.36 (0.93)	1.44 (1.11)	2.78 (1.36)	**1.34 (0.78)**	0.97
Vulkan Fast	1.43 (0.75)	2.54 (1.05)	3.25 (0.96)	2.66 (0.83)	3.28 (1.04)	1.23 (1.03)	1.55 (1.06)	1.48 (1.14)	2.32 (1.14)	1.76 (1.03)	1.0
Vulkan HQ	1.37 (0.72)	2.4 (1.0)	3.02 (0.89)	2.45 (0.76)	**3.04 (0.97)**	1.18 (0.98)	1.42 (0.97)	**1.29 (0.99)**	2.05 (1.0)	1.59 (0.93)	**0.92**
Shadow ray trace time (pre-transformed) [ms]											
HIPRT LBVH	6.82 (1.22)	11.91 (1.17)	8.41 (1.11)	1063.39 (69.05)	16.24 (1.16)	10.41 (1.22)	8.96 (1.25)	8.83 (1.23)	9.69 (1.34)	9.62 (1.15)	7.99
HIPRT PLOC	5.61 (1.0)	10.19 (1.0)	7.58 (1.0)	15.4 (1.0)	14.06 (1.0)	8.51 (1.0)	7.19 (1.0)	7.19 (1.0)	7.22 (1.0)	8.37 (1.0)	1.0
HIPRT SBVH	**4.84 (0.86)**	**5.19 (0.51)**	5.63 (0.74)	**7.56 (0.49)**	9.24 (0.66)	6.72 (0.79)	5.99 (0.83)	6.57 (0.91)	6.3 (0.87)	6.76 (0.81)	**0.75**
HIPRT Embree	4.87 (0.87)	5.92 (0.58)	**5.45 (0.72)**	8.05 (0.52)	**9.12 (0.65)**	**6.51 (0.76)**	**5.9 (0.82)**	**6.39 (0.89)**	**6.29 (0.87)**	**6.59 (0.79)**	**0.75**
Vulkan Fast	8.36 (1.49)	13.19 (1.29)	9.51 (1.25)	48.29 (3.14)	15.37 (1.09)	10.75 (1.26)	9.06 (1.26)	10.22 (1.42)	9.57 (1.33)	11.03 (1.32)	1.48
Vulkan HQ	6.99 (1.25)	12.67 (1.24)	8.17 (1.08)	11.04 (0.72)	13.62 (0.97)	9.22 (1.08)	7.68 (1.07)	7.26 (1.01)	8.14 (1.13)	8.87 (1.06)	1.06
Shadow ray trace time (two-levels) [ms]											
HIPRT LBVH	11.38 (1.07)	17.96 (1.19)	20.35 (1.08)	21.89 (1.72)	30.45 (1.14)	13.26 (1.19)	11.88 (1.26)	10.37 (1.07)	15.11 (1.18)	13.52 (1.17)	1.21
HIPRT PLOC	10.64 (1.0)	15.04 (1.0)	18.85 (1.0)	12.71 (1.0)	26.8 (1.0)	11.16 (1.0)	9.43 (1.0)	9.71 (1.0)	12.81 (1.0)	11.58 (1.0)	1.0
HIPRT SBVH	**9.52 (0.89)**	**13.75 (0.91)**	19.28 (1.02)	12.08 (0.95)	**25.45 (0.95)**	10.78 (0.97)	**9.05 (0.96)**	**8.6 (0.89)**	15.09 (1.18)	**9.81 (0.85)**	**0.96**
HIPRT Embree	10.06 (0.95)	18.62 (1.24)	21.01 (1.11)	**11.87 (0.93)**	27.46 (1.02)	**10.21 (0.91)**	9.66 (1.02)	9.31 (0.96)	15.12 (1.18)	10.64 (0.92)	1.02
Vulkan Fast	13.02 (1.22)	16.55 (1.1)	18.6 (0.99)	21.29 (1.68)	30.82 (1.15)	13.73 (1.23)	11.53 (1.22)	14.23 (1.47)	14.08 (1.1)	22.15 (1.91)	1.31
Vulkan HQ	11.5 (1.08)	15.68 (1.04)	**18.52 (0.98)**	18.92 (1.49)	28.15 (1.05)	12.68 (1.14)	10.4 (1.1)	12.4 (1.28)	**11.98 (0.94)**	21.23 (1.83)	1.19
Secondary ray trace time (pre-transformed) [ms]											
HIPRT LBVH	9.24 (1.2)	24.73 (1.1)	14.87 (1.12)	1083.84 (45.6)	27.41 (1.14)	12.97 (1.19)	11.95 (1.22)	11.99 (1.15)	12.1 (1.23)	12.21 (1.17)	5.61
HIPRT PLOC	7.69 (1.0)	22.54 (1.0)	13.24 (1.0)	23.77 (1.0)	24.06 (1.0)	10.89 (1.0)	9.8 (1.0)	10.44 (1.0)	9.86 (1.0)	10.47 (1.0)	1.0
HIPRT SBVH	6.26 (0.81)	**7.14 (0.32)**	7.99 (0.6)	**11.1 (0.47)**	12.57 (0.52)	7.49 (0.69)	**6.98 (0.71)**	8.11 (0.78)	7.41 (0.75)	7.77 (0.74)	**0.64**
HIPRT Embree	**6.25 (0.81)**	9.61 (0.43)	**7.56 (0.57)**	12.02 (0.51)	**12.22 (0.51)**	**7.38 (0.68)**	7.01 (0.72)	**7.44 (0.71)**	**7.36 (0.75)**	**7.13 (0.68)**	**0.64**
Vulkan Fast	9.22 (1.2)	27.27 (1.21)	12.26 (0.93)	60.2 (2.53)	27.15 (1.13)	11.12 (1.02)	12.18 (1.24)	12.04 (1.15)	10.75 (1.09)	12.28 (1.17)	1.27
Vulkan HQ	7.49 (0.97)	26.31 (1.17)	10.74 (0.81)	12.57 (0.53)	25.54 (1.06)	9.52 (0.87)	9.97 (1.02)	8.85 (0.85)	9.07 (0.92)	10.14 (0.97)	0.92
Secondary ray trace time (two-levels) [ms]											
HIPRT LBVH	15.36 (1.06)	39.09 (1.07)	33.22 (1.05)	30.84 (1.57)	54.82 (1.12)	14.61 (1.17)	17.42 (1.17)	14.63 (1.09)	20.22 (1.13)	16.35 (1.0)	1.14
HIPRT PLOC	14.48 (1.0)	36.67 (1.0)	31.75 (1.0)	19.65 (1.0)	48.9 (1.0)	12.48 (1.0)	14.92 (1.0)	13.48 (1.0)	17.87 (1.0)	16.33 (1.0)	1.0
HIPRT SBVH	**12.02 (0.83)**	**22.26 (0.61)**	**26.08 (0.82)**	16.83 (0.86)	**37.42 (0.77)**	11.05 (0.89)	**13.12 (0.88)**	**11.16 (0.83)**	20.76 (1.16)	**12.64 (0.77)**	**0.84**
HIPRT Embree	12.97 (0.9)	32.49 (0.89)	29.05 (0.91)	**15.48 (0.79)**	43.85 (0.9)	**10.84 (0.87)**	13.64 (0.91)	12.59 (0.93)	20.5 (1.15)	13.02 (0.8)	0.9
Vulkan Fast	20.75 (1.43)	39.39 (1.07)	36.48 (1.15)	28.18 (1.43)	69.26 (1.42)	21.27 (1.7)	19.02 (1.27)	15.48 (1.15)	19.05 (1.07)	30.95 (1.9)	1.36
Vulkan HQ	19.04 (1.31)	37.26 (1.02)	35.6 (1.12)	27.95 (1.42)	61.97 (1.27)	21.15 (1.69)	16.5 (1.11)	13.68 (1.01)	**16.44 (0.92)**	27.36 (1.68)	1.25
SAH cost (pre-transformed) [-]											
HIPRT LBVH	25.55 (1.24)	87.74 (1.12)	30.59 (1.23)	18.12 (1.02)	56.84 (1.15)	54.93 (1.34)	40.7 (1.32)	22.84 (1.16)	52.21 (1.29)	24.39 (1.26)	1.21
HIPRT PLOC	20.62 (1.0)	78.61 (1.0)	24.96 (1.0)	17.81 (1.0)	49.26 (1.0)	41.03 (1.0)	30.77 (1.0)	19.69 (1.0)	40.4 (1.0)	19.3 (1.0)	1.0
HIPRT SBVH	**18.79 (0.91)**	**26.1 (0.33)**	20.95 (0.84)	**17.03** (0.96)	**29.21 (0.59)**	31.75 (0.77)	25.21 (0.82)	16.4 (0.83)	36.34 (0.9)	15.71 (0.81)	**0.78**
HIPRT Embree	18.8 (0.91)	31.25 (0.4)	**20.66 (0.83)**	17.26 (0.97)	29.98 (0.61)	**31.69 (0.77)**	**24.88 (0.81)**	**15.39 (0.78)**	**35.93 (0.89)**	**15.59 (0.81)**	**0.78**

resolution of our 32-bit Morton codes. In such detailed scenes, the bottom levels contribute only negligibly to the SAH cost (due to division by the surface area of the scene bounding box) regardless of the actual BVH quality. To a certain extent, we can observe a similar tendency also for Vulkan Fast.

SAH Cost The SAH cost is a sum of the surface areas of leaves and internal nodes normalized by the surface area of the root. The hardware format makes the leaves fixed as it does not allow more than one triangle per leaf (except the triangle pairs), and thus, using different values of the SAH cost constants does not make a difference. SBVH and Embree are on par, achieving lower SAH costs than LBVH and PLOC. A noticeable exception is Bistro Interior, where the BVH has about 16% lower cost than Embree and 67% lower cost than PLOC, which is also reflected in the corresponding trace speed (see Figure 5).

Table 2. Average counts of tested nodes (internal nodes / instances / leaves) per ray for the HIPRT methods. Each leaf node contains one or two triangles (i.e., a triangle pair), which is given by the hardware-specific format.

	Trains	Bistro Int.	Hangar Ship	Opera House	Bistro Ext.	Museum	Sci-fi	Zero Day	Toy Shop	Yokohama
										
#triangles	836k	1207k	1235k	2512k	2829k	3650k	4809k	5165k	5212k	8217k
Primary rays (pre-transformed) [-]										
HIPRT LBVH	42 / 1 / 3	47 / 1 / 13	46 / 1 / 9	3866 / 1 / 14	54 / 1 / 8	47 / 1 / 5	53 / 1 / 5	36 / 1 / 6	46 / 1 / 3	50 / 1 / 4
HIPRT PLOC	31 / 1 / 3	30 / 1 / 13	36 / 1 / 10	105 / 1 / 15	42 / 1 / 9	29 / 1 / 5	35 / 1 / 5	22 / 1 / 5	32 / 1 / 3	38 / 1 / 5
HIPRT SBVH	31 / 1 / 3	22 / 1 / 2	25 / 1 / 4	54 / 1 / 11	35 / 1 / 5	22 / 1 / 3	27 / 1 / 3	26 / 1 / 3	26 / 1 / 2	27 / 1 / 3
HIPRT Embree	30 / 1 / 2	24 / 1 / 3	25 / 1 / 4	55 / 1 / 11	32 / 1 / 4	23 / 1 / 3	26 / 1 / 3	21 / 1 / 2	25 / 1 / 2	26 / 1 / 2
Primary rays (two-levels) [-]										
HIPRT LBVH	42 / 5 / 3	57 / 10 / 15	89 / 28 / 10	148 / 9 / 12	70 / 11 / 9	37 / 2 / 4	43 / 4 / 5	32 / 2 / 6	53 / 6 / 4	45 / 3 / 4
HIPRT PLOC	40 / 5 / 4	47 / 10 / 15	75 / 28 / 10	65 / 10 / 10	60 / 11 / 9	26 / 2 / 4	31 / 4 / 4	23 / 2 / 6	44 / 7 / 4	36 / 3 / 4
HIPRT SBVH	33 / 5 / 2	51 / 11 / 4	76 / 30 / 4	62 / 9 / 4	60 / 13 / 4	24 / 2 / 2	29 / 3 / 2	26 / 2 / 3	57 / 11 / 5	31 / 4 / 2
HIPRT Embree	33 / 5 / 2	65 / 14 / 10	84 / 32 / 6	58 / 8 / 8	60 / 13 / 6	23 / 2 / 3	30 / 4 / 3	28 / 2 / 3	59 / 12 / 5	31 / 4 / 3
Shadow rays (pre-transformed) [-]										
HIPRT LBVH	31 / 1 / 2	37 / 1 / 11	29 / 1 / 6	1854 / 1 / 4	40 / 1 / 8	32 / 1 / 4	31 / 1 / 4	27 / 1 / 3	30 / 1 / 4	29 / 1 / 3
HIPRT PLOC	23 / 1 / 2	28 / 1 / 10	22 / 1 / 5	55 / 1 / 4	31 / 1 / 8	23 / 1 / 4	22 / 1 / 4	20 / 1 / 3	22 / 1 / 3	22 / 1 / 3
HIPRT SBVH	21 / 1 / 2	23 / 1 / 4	21 / 1 / 4	29 / 1 / 4	28 / 1 / 5	20 / 1 / 3	21 / 1 / 3	22 / 1 / 3	20 / 1 / 3	19 / 1 / 3
HIPRT Embree	21 / 1 / 2	23 / 1 / 4	21 / 1 / 3	30 / 1 / 4	27 / 1 / 5	21 / 1 / 3	21 / 1 / 3	21 / 1 / 2	20 / 1 / 3	20 / 1 / 2
Shadow rays (two-levels) [-]										
HIPRT LBVH	28 / 3 / 2	42 / 6 / 11	36 / 8 / 5	64 / 3 / 3	47 / 7 / 8	29 / 2 / 3	26 / 3 / 3	25 / 2 / 3	28 / 3 / 3	27 / 2 / 3
HIPRT PLOC	25 / 3 / 2	28 / 4 / 8	29 / 7 / 5	29 / 3 / 3	39 / 7 / 8	22 / 2 / 3	21 / 2 / 3	21 / 2 / 3	25 / 4 / 4	22 / 2 / 3
HIPRT SBVH	24 / 3 / 1	35 / 6 / 4	36 / 10 / 3	30 / 4 / 3	40 / 7 / 4	22 / 3 / 2	22 / 3 / 2	21 / 2 / 2	25 / 4 / 2	21 / 2 / 2
HIPRT Embree	24 / 3 / 2	43 / 7 / 8	37 / 10 / 4	27 / 3 / 3	40 / 7 / 5	21 / 2 / 2	22 / 3 / 3	22 / 2 / 2	24 / 4 / 3	21 / 2 / 2
Secondary rays (pre-transformed) [-]										
HIPRT LBVH	40 / 1 / 4	57 / 1 / 38	47 / 1 / 12	2336 / 1 / 11	62 / 1 / 22	44 / 1 / 7	48 / 1 / 7	32 / 1 / 7	46 / 1 / 4	45 / 1 / 5
HIPRT PLOC	30 / 1 / 4	43 / 1 / 36	36 / 1 / 12	85 / 1 / 9	48 / 1 / 23	31 / 1 / 7	35 / 1 / 7	24 / 1 / 6	33 / 1 / 4	35 / 1 / 6
HIPRT SBVH	24 / 1 / 3	24 / 1 / 5	25 / 1 / 6	40 / 1 / 8	31 / 1 / 9	23 / 1 / 4	24 / 1 / 4	21 / 1 / 4	24 / 1 / 3	25 / 1 / 4
HIPRT Embree	24 / 1 / 3	25 / 1 / 8	26 / 1 / 5	41 / 1 / 8	30 / 1 / 7	24 / 1 / 4	24 / 1 / 4	19 / 1 / 3	24 / 1 / 3	25 / 1 / 3
Secondary rays (two-levels) [-]										
HIPRT LBVH	35 / 4 / 4	67 / 10 / 38	62 / 13 / 12	89 / 5 / 8	84 / 13 / 25	36 / 3 / 5	41 / 5 / 6	29 / 2 / 6	51 / 7 / 4	41 / 3 / 5
HIPRT PLOC	32 / 5 / 4	54 / 10 / 36	52 / 13 / 12	43 / 5 / 7	71 / 13 / 24	28 / 3 / 5	32 / 5 / 6	23 / 2 / 6	39 / 7 / 4	35 / 4 / 5
HIPRT SBVH	28 / 4 / 2	49 / 10 / 8	49 / 13 / 6	43 / 5 / 5	65 / 14 / 8	25 / 3 / 3	29 / 4 / 3	25 / 2 / 4	43 / 9 / 4	29 / 4 / 3
HIPRT Embree	28 / 4 / 3	57 / 11 / 20	51 / 14 / 8	39 / 5 / 5	68 / 14 / 14	25 / 3 / 3	29 / 4 / 4	26 / 3 / 4	43 / 9 / 4	29 / 4 / 4

Tested Nodes We also report hardware-independent statistics in a form of tested nodes in Table 2. The first number in each cell in the table is the number of tested internal nodes, the second number is the number of tested instances, and the last number is the number of tested leaves, where each leaf contains up to two triangles (i.e, a triangle pair).

SBVH achieves excellent results at the cost of higher construction times, which is not an issue for professional rendering as this cost is amortized through tracing many rays. Nonetheless, there are two other drawbacks. The first drawback is the high memory requirements given by the large number of allocated bins (1536B per task). The second drawback is inherent to the spatial splits. If we use any hit with spatial splits, the same hits are reported multiple times, which might be an issue for some applications.

HIPRT was also integrated into three advanced rendering systems to demonstrate its robustness and versatility: Blender Cycles [Blender Foundation 2023], PBRT-v4 [Pharr et al. 2023], and Radeon ProRender [AMD 2023a] (see Figures 1, 2, and 4).

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced HIPRT, a ray tracing framework tailored for professional rendering and general-purpose scientific computing on AMD GPUs. HIPRT achieves performance comparable with Vulkan, yet the HIPRT API is designed to be user-friendly and can be easily integrated with existing renderers. We proposed a new SBVH algorithm for massively parallel architectures, achieving high quality at the cost of higher construction times. We also addressed some of the design flaws of the existing APIs, such as motion blur or shader binding table. The goal of HIPRT is to offer a viable option to industrial renderers such as Blender Cycles on AMD GPUs rather than compete with other vendor-specific ray tracing engines.

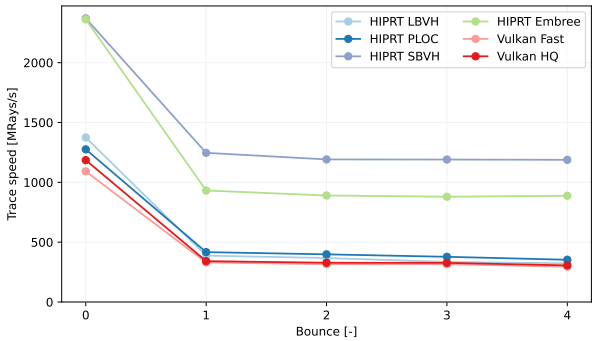

Fig. 5. Ray tracing performance of individual bounces for the Bistro Interior scene (pre-transformed). The 0-th bounce corresponds to primary rays and the other bounces to secondary rays.

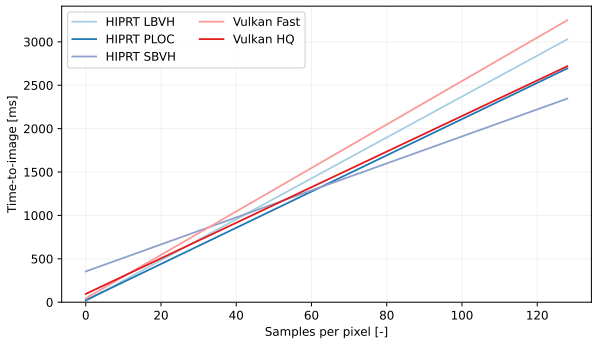

Fig. 6. Time-to-image of different samples counts for the Yokohama scene (pre-transformed). The offset in the origin corresponds to the build time. The high quality of SBVH outweighs the build overhead already around 64 samples per pixel.

There are a couple of interesting directions for future work. The results showed that 32-bit Morton codes are not sufficiently robust in some case. We will integrate the 64-bit extended Morton codes [Vinkler et al. 2017] to cover such cases. SBVH excels at scenes with oblong diagonal triangles. For custom primitives, SBVH splits only a bounding box, which is not optimal for thin oblong primitives such as curves. OptiX and Metal support specialized curves primitives. This allows split curves to get tighter bounding boxes, but it is restricted to a specific curve type. A more general solution could be to define a custom split function, leaving splitting up to the user. We plan to test HIPRT in use cases other than ray tracing, such as collision detection. Similarly to the custom intersection function, we could define a custom traversal function testing internal nodes. This would allow a user to traverse the BVH with another bounding box instead of the ray, which might be useful for the broad-phase in collision detection [Liu et al. 2010].

ACKNOWLEDGMENTS

We would like to thank other team members and colleagues for their valuable feedback and support on this project: Carsten Benthin, Richard Geslot, Sho Ikeda, ChihChen Kao, and Atsushi Yoshimura.

We also express our gratitude to the creators of the test scenes: Splash Fox (Daniel Bystedt), Landscape (Laubwerk), Edo (Shunsuke Nakajo), Moana Island (Disney Animation), Hair (Cem Yuksel), Trains (Denis Rutkovsky), Bistro (Amazon Lumberyard), Hangar Ship (Luxsoft), Opera House (ArtcoreStudios), Museum (ArtcoreStudios), Sci-fi (Jonathon Frederick), Zero Day (Beeple), Toy Shop (Luxsoft), and Yokohama (Kyrylo Sibiriakov).

Copyright Notice and Trademarks

REFERENCES

Andy Adinets and Duane Merrill. 2022. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs. *arXiv e-prints* (2022), arXiv:2206.01784.

Attila Áfra and László Szirmay-Kalos. 2014. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum* 33, 1 (2014), 129–140.

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics*. 145–149.

AMD. 2022. Orochi. https://gpuopen.com/orochi/

AMD. 2023. GPURT. https://github.com/GPUOpen-Drivers/gpurt.

AMD. 2023a. Radeon ProRender. https://www.amd.com/en/technologies/radeon-prorender

AMD. 2023b. RDNA3 Instruction Set Architecture. https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf

Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. In *Proceedings of Computer Graphics and Visual Computing*.

Apple. 2023. Metal Shading Language Specification. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

Rasmus Barringer and Tomas Akenine-Möller. 2013. Dynamic Stackless Binary Tree Traversal. *Journal of Graphics Tools* 2, 1 (2013), 38–49.

Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. 2022. PLOC++ : Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. In *Proceedings of High-Performance Graphics*.

Carsten Benthin, Sven Woop, Ingo Wald, and Attila Áfra. 2017. Improved Two-Level BVHs using Partial Re-Braiding. In *Proceedings of High-Performance Graphics*.

Nikolaus Binder and Alexander Keller. 2016. Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time. In *Proceedings of High-Performance Graphics*. 41–50.

Blender Foundation. 2023. Cycles. https://docs.blender.org/manual/en/latest/render/cycles/index.html

Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics* 37, 3 (2018).

Holger Dammertz and Alexander Keller. 2008. Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance. In *Proceedings of Symposium on Interactive Ray Tracing*. 155–158.

Disney. 2016. Moana Island Scene. https://disneyanimation.com/resources/moana-island-scene/.

DreamWorks. 2023. MoonRay Production Renderer. https://openmoonray.org/

Manfred Ernst and Sven Woop. 2011. Ray Tracing with Shared-Plane Bounding Volume Hierarchies. *Journal of Graphics, GPU, and Game Tools* 15, 3 (2011), 141–151.

Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. 2016. Parallel Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. 21–30.

Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and Faster HLBVH with Work Queues. In *Proceedings of High-Performance Graphics*. 59–64.

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics* 37, 3 (2018).

Iliyan Georgiev and Philipp Slusallek. 2008. RTfact: Generic concepts for flexible and high performance ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*. 115–122.

Michael Guthe. 2014. Latency Considerations of Depth-first GPU Ray Tracing. In *Proceedings of Eurographics (Short Papers)*.

Eric Haines and Tomas Akenine-Möller (Eds.). 2019. *Ray Tracing Gems*. Apress. http://raytracinggems.com.

Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. 2013. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings of Spring Conference on Computer Graphics*. 7–12.

Jakub Hendrich, Daniel Meister, and Jiří Bittner. 2017. Parallel BVH Construction Using Progressive Hierarchical Refinement. *Computer Graphics Forum (Proceedings of Eurographics)* 36, 2 (2017), 487–494.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 41, 4 (2022).

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of High-Performance Graphics*. 33–37. https://research.nvidia.com/publication/maximizing-parallelism-construction-bvhs-octrees-and-k-d-trees

Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics*. 89–100.

Khronos Group. 2020. Vulkan Ray Tracing Extensions Specification. https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release

Samuli Laine. 2010. Restart Trail for Stackless BVH Traversal. In *Proceedings of High-Performance Graphics*. 107–111.

Samuli Laine and Tero Karras. 2015. Apex Point Map for Constant-Time Bounding Plane Approximation. In *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Alexander Lier, Marc Stamminger, and Kai Selgrad. 2018. CPU-style SIMD Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics*. 7:1–7:4.

Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. 2010. Real-Time Collision Culling of a Million Bodies on Graphics Processing Units. *ACM Transactions on Graphics* 29, 6 (2010).

Daniel Meister and Jiří Bittner. 2018. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics* 24, 3 (2018), 1345–1353.

Daniel Meister and Jiří Bittner. 2022. Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing. *Journal of Computer Graphics Techniques (JCGT)* 11, 4 (2022), 1–19.

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics)* 40, 2 (2021). https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662

Daniel Meister, Atsushi Yoshimura, and Chih-Chen Kao. 2023. GPU Programming Primitives for Computer Graphics. In *ACM SIGGRAPH Asia 2023 Courses (SIGGRAPH Asia 2023)*.

Duane Merrill and Andrew Grimshaw. 2011. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters* 21, 2 (2011), 245–272.

Microsoft. 2020. DirectX Raytracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

Guy Morton. 1966. *A Computer Oriented Geodetic Database and a New Technique in File Sequencing*. Technical Report.

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38, 6 (2019).

Steven Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics* 29, 4 (2010), 66:1–66:13.

Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: generating renderers without writing a generator. *ACM Transactions Graphics* 38, 4 (2019).

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically Based Rendering: From Theory to Implementation (3rd ed.)* (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

André Susano Pinto. 2010. Adaptive Collapsing on Bounding Volume Hierarchies for Ray-Tracing. In *Proceedings of Eurographics (Short Papers)*.

Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. 2009. Object Partitioning Considered Harmful: Space Subdivision for BVHs. In *Proceedings of High-Performance Graphics*. 15–22.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of the High-Performance Graphics*. 7–13.

K. Vaidyanathan, S. Woop, and C. Benthin. 2019. Wide BVH Traversal with a Short Stack. In *Proceedings of High-Performance Graphics*.

Marek Vinkler, Jiří Bittner, and Vlastimil Havran. 2017. Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction. In *Proceedings of High-Performance Graphics*.

Ingo Wald, Carsten Benthin, and Philipp Slusallek. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of Symposium on Parallel and Large-Data Visualization and Graphics*. 77–86.

Ingo Wald, Nate Morrical, and Eric Haines. 2024. OWL: A Node Graph "Wrapper" Library for OptiX 7. https://github.com/owl-project/owl

Ingo Wald, Sven Woop, Carsten Benthin, Gregory Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33 (2014).

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Proceedings of High-Performance Graphics*. 4:1–4:13.

Shaokun Zheng, Zhiqian Zhou, Xin Chen, Difei Yan, Chuyan Zhang, Yuefeng Geng, Yan Gu, and Kun Xu. 2022. LuisaRender: A High-Performance Rendering Framework with Layered and Unified Interfaces on Stream Architectures. *ACM Transactions on Graphics* 41, 6 (2022).