RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms

Arsène Pérard-Gayot
Computer Graphics Lab
Saarland University
Saarbrücken, Germany
perard@cg.uni-saarland.de

Martin Weier
Institute of Visual Computing
Bonn-Rhein-Sieg University
Sankt Augustin, Germany
martin.weier@h-brs.de

Richard Membarth
Agents and Simulated Reality
DFKI
Saarbrücken, Germany
richard.membarth@dfki.de

Philipp Slusallek
Agents and Simulated Reality
DFKI
Saarbrücken, Germany
philipp.slusallek@dfki.de

Roland Leißa
Compiler Design Lab
Saarland University
Saarbrücken, Germany
leissa@cs.uni-saarland.de

Sebastian Hack
Compiler Design Lab
Saarland University
Saarbrücken, Germany
hack@cs.uni-saarland.de

Abstract

In order to achieve the highest possible performance, the ray traversal and intersection routines at the core of every high-performance ray tracer are usually hand-coded, heavily optimized, and implemented separately for each hardware platform—even though they share most of their algorithmic core. The results are implementations that heavily mix algorithmic aspects with hardware and implementation details, making the code non-portable and difficult to change and maintain.

In this paper, we present a new approach that offers the ability to define in a functional language a set of conceptual, high-level language abstractions that are optimized away by a special compiler in order to maximize performance. Using this abstraction mechanism we separate a generic ray traversal and intersection algorithm from its low-level aspects that are specific to the target hardware. We demonstrate that our code is not only significantly more flexible, simpler to write, and more concise but also that the compiled results perform as well as state-of-the-art implementations on any of the tested CPU and GPU platforms.

CCS Concepts • **Computing methodologies → Computer Graphics**; **Ray Tracing**; • **Software and its engineering → Domain-Specific Languages**;

Keywords Computer Graphics, Ray Tracing, Functional Programming, Domain-Specific Languages

GPCE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136044

ACM Reference Format:
Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. 2017. RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136040.3136044

1 Introduction

Image synthesis algorithms determine visibility by tracing millions of rays per frame and finding intersections with the geometric primitives in the scene. Spatial index or acceleration structures are used to speed up this process by quickly eliminating many unnecessary intersection tests. They organize the scene geometry into a tree or a grid by subdividing it into smaller subsets. The task of a *traversal algorithm* then is to traverse this data structure and perform any necessary intersection tests *as fast as possible.*

Therefore, a lot of research has focused on optimizing this core algorithm, specifically by taking advantage of modern hardware architectures that provide parallelism on multiple levels. A typical CPU has multiple cores, each of them supporting vector instructions that operate on multiple data elements in parallel (SIMD). Modern GPUs are equipped with hundreds of parallel execution units allowing a single instruction to be executed in parallel with multiple threads (SIMT). These architectural differences have led to various implementations of ray traversal algorithms that are closely tied to a specific hardware and execution model.

For example, on the CPU, Bounding Volume Hierarchy (BVH) traversal algorithms benefit from SIMD instructions by using packet tracing [42] or single-ray tracing and wide BVHs [41]. On GPUs, these approaches are not so attractive: Single-ray tracing with standard BVHs turns out to be more efficient [1].

Conceptually, the traversal algorithms for CPUs and GPUs are identical. Hence, it would be desirable to implement the algorithm *once* in an abstract way and later *refine* only the hardware-specific parts for a new architecture in order

to save effort. In software that is not performance-critical, abstraction is usually achieved by procedural abstraction, object-orientation, and in particular dynamic method dispatch. However, some of these techniques incur a significant runtime overhead that even modern compilers are often unable to remove *reliably*. As a result, this optimization process today is usually performed manually by experts that understand both the algorithmic *as well as* the hardware-specific aspects. As a consequence, the resulting code is no longer abstract: It uses concrete instead of abstract data types and is polluted by the idiosyncrasies of the target hardware. Consequently, it is not portable (at least not performance portable), hard to maintain, debug, and extend.

Consider Figure 1: It shows three different implementations of the part of BVH traversal that is responsible for the intersection of the child nodes. The ray-box intersection routine has been highlighted in blue.

The examples on the left and middle are taken from state-of-the-art implementations for CPUs and GPUs [1, 43], and the right code snippet contains our high-level description of that code. In Figure 1a and Figure 1b, there is no clean separation between the code that intersects the bounding boxes and the one that updates the stack.

In contrast, our generic description captures the essence of the algorithm and isolates the three following concepts: first, the iteration over the set of child nodes, whose size may be greater than two, depending on the acceleration structure; second, the intersection of the ray, or packet of rays with each box; and third, the process of pushing any of the corresponding nodes onto the stack, in sorted order. This is achieved using a higher-order function, `hit`, highlighted in red in Figure 1c, that is provided by the implementation of the iteration abstraction `iterate_children` and performs the stack operations. The implementation of `hit` depends on the concrete acceleration structure and has to be carefully tuned for the respective target hardware.

Hence, when mapping the generic code to a specific architecture the programmer *instantiates* the traversal algorithm to a respective target platform by supplying implementations for this and the other target-specific abstractions. To be as efficient as hand-tuned code it is mandatory that all "calls" to these abstractions are reliably removed and function arguments are propagated at compile time. Standard compiler optimizations generally do not guarantee this for various reasons. In our setting, we use a programming language that supports partial evaluation—the symbolic evaluation of a part of the program before its actual execution—to *reliably* remove all overhead of our abstractions. The result of the partially evaluated kernel is code that looks and performs as if it had been manually refined and optimized to each particular hardware architecture—simply by recompiling with the specific "library" of hardware-specific refinements.

In this paper, we apply these recent results of research on meta-programming to the implementation of high-performance traversal algorithms. We show that higher-order functions *combined* with partial evaluation are sufficient to generate high-performance implementations from a generic traversal algorithm. We demonstrate that this algorithm can be easily modified to support common variants such as early ray termination or transparency.

Contributions

In summary, this paper makes the following contributions:

- We show how to use higher-order functions to abstract ray traversal (Section 3). We separate the traversal into two parts: The high-level description of the traversal loop and its associated mapping to different hardware architectures. The former is written in textbook-like style while the latter contains the architecture-specific details.
- We give examples of possible mappings for GPUs and CPUs that exploit low-level features of the underlying hardware (Section 4). These mappings particularly target CUDA and CPUs with SIMD instruction sets.
- We demonstrate that our approach is competitive in terms of performance: It is on a par or even outperforms state-of-the-art, highly-optimized, and hand-tuned implementations for GPUs as well as CPUs (Section 6.1).
- Using objective software complexity measures we also show that this approach is significantly more flexible, simpler, and easier to implement (Section 6.2).

2 Related Work

Ray Tracing Early ray tracers traced a single ray for each CPU or node in a cluster [12, 22, 30]. Wald et al. [42] demonstrated how to leverage SIMD hardware by vectorized intersection routines that operate on packets of rays. Packet traversal was later further improved with frustum culling [6, 34] and other techniques that increase SIMD coherence [7, 13, 41]. Another category of traversal algorithms work on streams of rays [5, 39]. These algorithms extract coherence by filtering a group of rays during traversal in order to take best advantage of the SIMD units.

GPUs have become more and more programmable and are now suitable for general purpose computations. The parallel nature of ray tracing has led to many attempts to make it run efficiently on GPUs [e.g. 15]. Early ray tracing systems for BVHs on the GPU used packetized traversal [17, 20], while current state-of-the-art approaches trace single rays by maintaining a traversal stack for each ray, as in the work of Aila and Laine [1]. The latter work and its extension [2] also outline the importance of the traversal loop shape, the scheduling, and the optimization of memory accesses.

As implementing efficient ray tracing routines demands much expertise and machine knowledge, several frameworks have been developed. One of the first attempts was OpenRT [40]: It contains a simple OpenGL-like API abstracting over a set of optimized SSE routines. Only its core functionality is vectorized, and shading is done in single-ray fashion. The

```
for (unsigned i = 0; i < 4; i++) {
  const NodeRef child = node->children[i];
  if (unlikely(child == BVH4::emptyNode))
    break;

  avxf lnearP;
  const avxb lhit = node->intersect8(i,
    org, rdir, org_rdir,
    ray_tnear, ray_tfar, lnearP);
  if (likely(any(lhit))) {
    const avxf childDist =
      select(lhit, lnearP, inf);

    sptr_node++;
    sptr_near++;

    if (any(childDist < curDist)) {
      *(sptr_node-1) = cur;
      *(sptr_near-1) = curDist;
      curDist = childDist;
      cur = child;
    } else {
      *(sptr_node-1) = child;
      *(sptr_near-1) = childDist;
    }
  }
}
```

(a) Sample from a CPU implementation

```
const float c0min = spanBegin(c0lox, c0hix,
  c0loy, c0hiy, c0loz, c0hiz, tmin);
const float c0max = spanEnd(c0lox, c0hix,
  c0loy, c0hiy, c0loz, c0hiz, hitT);
const float c1min = spanBegin(c1lox, c1hix,
  c1loy, c1hiy, c1loz, c1hiz, tmin);
const float c1max = spanEnd(c1lox, c1hix,
  c1loy, c1hiy, c1loz, c1hiz, hitT);
bool swp = (c1min < c0min);
bool traverseChild0 = (c0max >= c0min);
bool traverseChild1 = (c1max >= c1min);

if (!traverseChild0 && !traverseChild1) {
  nodeAddr = *(int*)stackPtr;
  stackPtr -= 4;
} else {
  nodeAddr = (traverseChild0) ?
      cnodes.x : cnodes.y;

  if (traverseChild0 && traverseChild1) {
    if (swp) swap(nodeAddr, cnodes.y);
    stackPtr += 4;
    *(int*)stackPtr = cnodes.y;
  }
}

if (nodeAddr < 0 && leafAddr >= 0) {
  leafAddr = nodeAddr;
  nodeAddr = *(int*)stackPtr;
  stackPtr -= 4;
}
```

(b) Sample from a GPU implementation

```
for min, max, hit in
    iterate_children(node, stack) {
  intersect_ray_box(org, idir,
                    tmin, t,
                    min, max, hit);
}
```

(c) Sample from our traversal

Figure 1. Sample implementations of the part of BVH traversal responsible for intersecting the children of the current node and pushing the next nodes on the stack. The left and center snippets have been extracted from state-of-the-art implementations.

rendering API RTfact [16] later tried to overcome this issue by using template meta-programming in C++. In RTfact, templated classes for traversal and shading provide a framework on top of which renderers are instantiated. However, templates are hard to use, maintain, and extend. They clutter code with uncountable template annotations and often generate unfathomable error messages when used improperly. In addition, a lot of specialized and complicated code must be written to support a wide variety of SIMD widths and hardware architectures. The current state-of-the-art CPU ray tracing system Embree [43] supports different BVH structures and packet sizes. The core ray tracing kernels of Embree are written and optimized by hand with vector intrinsics for performance reasons. The Embree example renderer uses `ispc` [32], a compiler for a C-like programming language that hides the complexity of vector intrinsics and data types. However, `ispc` cannot be used to build complex abstractions, since it lacks support for more advanced language features like polymorphism or higher-order functions. OptiX [31] is built on top of CUDA. It follows a single-ray approach and internally uses the kernels from Aila et al. [2]. A specific compiler merges the kernels written with OptiX and creates a *megakernel* out of them. The resulting program can only run on NVIDIA GPUs or on CPUs via a PTX to x86 compiler. However, the CPU backend does not perform nearly as well as Embree. Radeon-Rays[1] (previously FireRays) is another GPU ray tracing framework based on OpenCL which offers a simple API for scene management, acceleration structure construction, and ray traversal. Unfortunately, their algorithm is not as optimized as the traversal from Aila et al. [2] and is consequently slower. As we see, the trend is towards incorporating compiler technology in high-performance ray tracing. In this context, compilers have two major uses: Perform transformations that would otherwise be manual, and explore different variants [36].

Andersen [4] uses C-Mix [3]—a partial evaluator for C—in order to specialize a ray tracer with respect to objects and light sources in the scene. Depending on the specialized scene, the achieved speed up is between 1.5x-3.x. However, the evaluated scenes only consist of a few objects. So this approach does not scale with complex scenes containing tens of thousands of triangles.

Domain-specific Languages In other areas like computer vision or high-performance computing, domain-specific languages (DSLs) have been successfully used in order to gain performance from a straightforward implementation. DSLs can be either implemented from scratch like Diderot [11] or embedded into a *host language* like C++ or Scala. When embedding DSLs one can reuse the lexer, parser, and type system of the host language. Carette et al. [9] and Hudak [21] lay the foundation of embedding a typed language by ordinary functions instead of object terms. Hofer et al. [19] picked up this idea and carried it to the Scala world while emphasizing modularity and the ability to abstract over and compose semantic aspects. Rompf and Odersky [35] coined

[1] http://gpuopen.com/gaming-product/radeon-rays

the term Lightweight Modular Staging (LMS) and paved the way for performance-oriented embedded DSLs [10] like OptiML [38] or Liszt [14]. LMS does not rely on explicit staging capabilities of the host language Scala. Instead, executing the host program constructs a second domain-specific program representation like Delite [8]. For this reason, this kind of embedding is called *deep embedding*. Examples include Spiral in Scala [29] that implements a generator for DSP algorithms in LMS, or Array Building Blocks [27] and Halide [33] that leverage a similar staging mechanism to construct the actual program representation with C++ as host language for vector programming and image processing respectively. HIPA^{cc} [25] and SYCL[2] on the other hand, are *shallowly* embedded DSLs in C++. In contrast to deeply embedded DSLs, HIPA^{cc} and SYCL programs can be compiled with an unmodified C++ compiler. However, a *modified* C++ compiler, that directly manipulates the program representation, is needed in order to achieve performance. Our work uses shallow embedding. But instead of extending the compiler of the host language, we use partial evaluation in order to eliminate higher-order functions at compile-time [23, 24].

3 Abstractions for Ray Traversal

RaTrace, our high-performance traversal implementation, is written in Impala [23], an imperative and functional language that borrows from the language Rust. In contrast to Rust, Impala integrates a partial evaluator that allows us to remove abstractions at compile time.

3.1 Zero-cost Abstractions

Most of the Impala code should be self explanatory but we give a brief overview of non-standard features below.

Impala supports SIMD vectors. A vector of four floats is denoted by the type `simd[float * 4]`.

A `for` loop in Impala is merely syntactic sugar to call a higher-order function. The construct

```
for var1, var2 in iter_func(arg1, arg2) { /* ... */ }
```

translates to

```
iter_func(arg1, arg2, |var1, var2| { /* ... */ });
```

The body of the `for` loop and the iteration variables constitute an anonymous function `|var1, var2| { /* ... */ }` that is passed to `iter_func` as the last argument. It can then be called from within the function `iter_func`.

This can nicely be used to separate the *high-level description* of an algorithm from its hardware-specific *mapping*, which often also determines the *schedule* of how to best apply operations on some data structure. For example, consider the following high-level image processing abstraction that scales each pixel of an image:

```
for x, y in iterate(img) {
  img(x, y) = img(x, y) * 0.5f;
}
```

[2] https://www.khronos.org/sycl

A simple mapping for the `iterate` function onto the CPU then applies this operation sequentially over all pixels of the image.

```
fn iterate(img: Image, body: fn(int, int)->()) -> () {
  for y in range(0, img.height) {
    for x in range(0, img.width) {
      body(x, y);
    }
  }
}
```

More optimized CPU mappings (e.g. ones that use vectorization or loop blocking) can be implemented accordingly. A GPU implementation of `iterate` would use SIMT parallelism to process vectors of pixels. Impala offers built-in higher-order functions to express that a piece of code (given as a closure) is to be executed on a GPU. For example, the following implementation of `iterate` leverages the built-in function `cuda` to execute the kernel on the GPU via CUDA:

```
fn iterate(img: Image, body: fn(int, int)->()) -> () {
  let grid  = (img.width, img.height);
  let block = (32, 4);
  cuda(grid, block, || {
    let x = tid_x() + bid_x()*bdim_x();
    let y = tid_y() + bid_y()*bdim_y();
    body(x, y);
  });
}
```

Usually, calling a function that binds variables from its environment requires to build a closure. By default, the Impala compiler partially evaluates all calls to higher-order functions, which removes closures for non-recursive calls. Using a novel optimization called *lambda mangling* [24], the evaluator also eliminates closures for tail-recursive calls.

This allows the programmer to cleanly *separate* the code between the high-level, generic algorithm and low-level, hardware-specific mappings using higher-order functions. Impala's partial evaluator combines these code parts by removing the overhead of closure allocation and function calls entirely.

In the case of RaTrace, all higher-order calls are either non-recursive or tail-recursive. Hence, the default partial evaluation removes all closures. Additionally, the programmer can force partial evaluation by annotating a call with `@f(/*...*/)` [23]. RaTrace uses these additional explicit partial evaluation annotations to unroll loops, force the inlining of functions, or specialize code.

3.2 Traversal Algorithm

A BVH is an acceleration structure for ray tracing, which consists of a n-ary tree (n is usually 2 or 4) in which every node is associated to a *bounding volume* (a box, typically) and every leaf contains a list of primitives. An example of such a BVH is given in Figure 2.

Given a ray, a traversal algorithm traverses this tree to find the closest intersection by recursively intersecting the ray with the bounding box of each node. The two children of the node are processed only if there is an intersection, which eliminates many ray-primitive tests compared to a

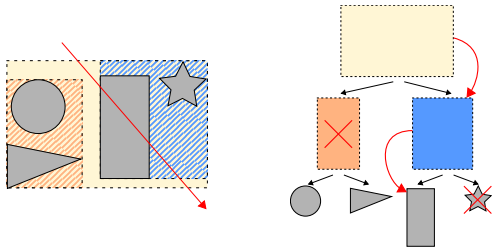

Figure 2. A scene made of 4 primitives (left) and one possible BVH (right). A ray and its traversal path are drawn in red.

brute-force method. Different variants of this basic algorithm have been proposed to increase performance on different architectures, but the core idea behind them remains the same, as will be shown in the next section.

3.3 Traversal Abstractions

Our traversal implementation RaTrace builds on a careful study of the algorithms used by Embree and Aila et al. [2, 43]. Both implementations share several common aspects:

- The *outermost loop*, which iterates over the set of rays. In Embree, this iteration over rays is implicit: Vectorization can be seen as an iteration over N rays (N being the SIMD width).
- The *initialization*, in which the stack and traversal variables are allocated.
- The *traversal loop*, which terminates when the stack is empty.
- The *innermost loop*, which either iterates over internal nodes in the case of Embree or leaves for Aila's code. Since our experiments have shown that the GPU is more sensitive to control flow instructions, it is acceptable to iterate over leaves for both the CPU and GPU implementations.
- The *intersection routines* that can use precomputed data to increase performance.

But they also differ considerably in several aspects:

- The *acceleration structure*: Embree uses a 4-wide BVH; Aila et al. use a standard binary BVH.
- The *vectorization*: Embree traces packets of 8 rays; Aila et al. use single rays.
- The *data layout*: Rays, nodes, and triangles are loaded and stored in a way that maximizes performance for each architecture.

This analysis suggests that our traversal algorithm must support BVHs of different arity and that only the acceleration structure, vectorization, and data layout should be kept separate for each implementation.

A version of our generic and high-level traversal code is given in Listing 1. For the sake of presentation, small parts of the code that are not relevant for our discussion are omitted. Within the iteration over all rays the algorithm proceeds as follows: First, a stack is allocated and the traversal variables are initialized. Next, the root node is pushed onto the top of the stack and the traversal starts. We assume that the root of the BVH is not a leaf, hence we begin by iterating over the children of the current node in **Step 1**. Any children that are intersected by the current ray are pushed onto the stack. The order in which they are pushed is defined by an architecture-specific heuristic provided in the mapping. The top of the stack now contains the next node to intersect: If it is a leaf, we iterate over its triangles and record any intersection in **Step 2**, otherwise we return to **Step 1**.

```
for tmin, tmax, org, dir, record_hit in
    iterate_rays(ray_list, hit_list, ray_count) {
  // The ray is defined as org + t * dir
  // with t in between tmin and tmax

  // Allocate a stack for the traversal
  let stack = allocate_stack();

  // Initialize traversal variables
  let idir = 1.0f / dir;
  let mut t = tmax;
  let mut u = 0.0f;
  let mut v = 0.0f;
  let mut tri_id = -1;

  stack.push_top(root, tmin);

  // Traversal loop
  while !stack.is_empty() {

    // Step 1: Intersect children and update the stack
    for min, max in iterate_children(t, stack) {
      intersect_ray_box(org, idir, tmin, t, min, max)
    }

    // Step 2: Intersect the leaves
    while is_leaf(stack.top()) {
      let leaf = stack.top();

      for id, tri in iterate_triangles(leaf, tris) {
        let (mask, t0, u0, v0) =
          intersect_ray_tri(org, dir, tmin, t, tri);

        t      = select(mask, t0, t);
        u      = select(mask, u0, u);
        v      = select(mask, v0, v);
        tri_id = select(mask, id, tri_id);
      }

      stack.pop();
    }
  }

  record_hit(tri_id, t, u, v);
}
```

Listing 1. Our generic traversal loop.

The intersection routines, highlighted in red, can easily be changed without impacting the core traversal algorithm. The types of their arguments are generic, so they can operate on single rays or packets of rays using SIMD units. Their return value is the result of the intersection: The entry and exit distance in the case of a ray-box intersection, or a Boolean flag—set if a triangle is hit—along with the hit distance and barycentric coordinates in the case of a ray-triangle intersection. The triangle intersection routine additionally uses an abstract data structure for the input primitive, which allows us to use the original triangle data or use precomputed data [44].

In **Step 1**, the result of the ray-box intersections are directly forwarded to `iterate_children`. Thus, it can push or

pop nodes depending on their relative intersection distances and specific mappings can make different decisions for the child traversal order.

In our implementation, both the CPU and the GPU version share the same intersection routines. We use the slab test [22] to compute ray-box intersections and the Möller-Trumbore intersection algorithm [26] to compute the ray-triangle intersections.

The traversal stack stores the node identifiers as well as the corresponding entry distance along the ray (the exit distance is discarded). These distances are used to cull nodes when an intersection with a triangle has been found.

Note that Listing 1 corresponds to a textbook-like algorithmic description of ray traversal. Still, the chosen abstractions allow us to map this generic code to highly optimized, hardware-specific code, as shown below.

4 Mappings to Different Architectures

The mapping of the traversal algorithm to each architecture requires a different implementation of our three hardware-specific abstractions: `iterate_rays`, `iterate_ children`, and `iterate_triangles`. These functions take care of all the low-level details and ensure that the traversal routine takes advantage of every available hardware feature.

4.1 CPU Mapping

Our CPU mapping uses 4-wide BVHs. For each node, we store the bounding boxes of the four children in a structure-of-arrays layout and their index in the array of nodes (Listing 2). A special value of zero for the child index corresponds to an empty node; negative values represent leaves and point to the triangle array. Instead of storing the number of primitives in the node structure, sentinels are stored in the triangle array to indicate the end of a leaf, which minimizes the size of the node structure.

```
struct Node {
  min_x: [float * 4],
  min_y: [float * 4],
  min_z: [float * 4],
  max_x: [float * 4],
  max_y: [float * 4],
  max_z: [float * 4],
  children: [int * 4]
}
```

Listing 2. Structure of a BVH node on the CPU.

The CPU mapping for the traversal loop uses packets of 8 rays for best use of the AVX2 SIMD instruction set. This means that in the generic code the type of `org`, `dir`, `tmin`, `tmax` as returned by `iterate_rays`, and all the traversal variables containing per-ray data are actually inferred by the compiler to be SIMD vectors. For instance, the compiler will infer that the type of `tmin` in Listing 1 is `simd[float * 8]`.

The core part of the mapping is the `iterate_children` function (Listing 3): This is where the nodes are loaded and the decision is made to continue the traversal with one or more children.

```
fn iterate_children(t: simd[float * 4],
                    stack: Stack,
                    body: IntersectionFn) -> () {
  let (node, tmin) = stack.top();
  stack.pop();

  // If the current distance for all the rays in the
  // packet is smaller than the entry distance of this
  // node, then discard it
  if all(t < tmin) { return() }

  // Iterate over the children of this node
  for i in unroll(0, 4) {
    // Are this child and the following empty ?
    if node.children(i) == 0 { break() }

    // Get the bounding box for this child
    let min = vec3(node.min_x(i),
                   node.min_y(i),
                   node.min_z(i));
    let max = vec3(node.max_x(i),
                   node.max_y(i),
                   node.max_z(i));

    // Call the intersection function (the loop body)
    // This returns the entry and exit distance
    let (tentry, texit) = body(min, max);

    let t = select(texit>=tentry,tentry,flt_max);
    // Is the intersection valid ?
    if any(texit >= tentry) {
      // Yes, then push the nodes so that the
      // closest one is intersected first
      if any(stack.tmin() > t) {
        stack.push_top(node.children(i), t)
      } else {
        stack.push(node.children(i), t)
      }
    }
  }
}
```

Listing 3. CPU mapping of the `iterate_children` function.

In `iterate_children`, we first pop a node from the stack and test if its associated intersection distance is less than any recorded distance along the packet of rays using the function `any` that will return true if the condition holds for *any* lane when SIMD types are used. If this is the case, we iterate over the children of the node (this mapping uses a BVH4 with four children per node) and for each of them we intersect the associated bounding box with the packet. The intersection is computed by calling the function `body` that is passed to `iterate_children`. Thanks to the special `for` loop syntax of Impala, this integrates nicely with the generic traversal code: The function `body` is actually the body of the `for` loop in **Step 1** of the traversal (Listing 1).

The remaining part of the CPU mapping is straightforward: `iterate_rays` loads a set of 8 rays and returns them in SIMD variables, while `iterate_triangles` loads the triangles from possibly hardware-specific layouts and performs the intersection computation provided in the loop body.

4.2 GPU Mapping

The GPU mapping traverses a standard BVH in single-ray fashion and thus uses a different memory layout. A BVH node contains the bounding box of the two children along with their indices in the array of nodes (Listing 4). The same idea as in the CPU mapping is used here: Negative values

are interpreted as leaves and give the index into the triangle array.

```
struct BBox {
  lo_x: float,
  hi_x: float,
  lo_y: float,
  hi_y: float,
  lo_z: float,
  hi_z: float
}

struct Node {
  left_bb: BBox,
  right_bb: BBox,
  left_child: int,
  right_child: int,
  padding: [int * 2] // 16-byte alignment
}
```

Listing 4. Structure of a BVH node on the GPU.

In order to improve GPU performance, we use 128-bit vector loads when loading sets of rays, triangles, and BVH nodes from global memory. We also exploit the caches in recent GPU architectures by using read-only memory loads, using the hardware-specific intrinsic `ldg`, exposed by the compiler. This gives a small speedup of a few percent in our test scenarios.

The GPU implementation of `iterate_children` is presented in Listing 5. **Step 1** first loads the BVH node using vector loads and **Step 2** shuffles the values in those vectors to build a bounding box. Finally, we call the intersection function for both child bounding boxes, sort the nodes by their distance along the ray, and push them onto the stack. The intersection function is called `body` because it contains the body of the `for` loop in **Step 1** of Listing 1.

The traversal variables (e.g., `org`, `dir`, etc.) are not vectors in the GPU mapping: The compiler infers that they are of type `float` from the GPU mapping of the `iterate_rays` function.

In this version of `iterate_children`, we do not immediately pop a node from the stack. This is because the top of the stack is stored as a separate variable: If we intersect only a single child, it is more efficient to simply replace the contents of that variable (this is what the `set_top` function does). This seemingly small optimization has an important impact on performance, mainly because it replaces a few reads and writes to memory by reads and writes to registers.

The same key ideas are applied in `iterate_rays` and `iterate_triangles`: We use vector loads and `ldg` to get the data into registers. Their implementation is otherwise very straightforward.

For testing purposes, we evaluated the use of persistent threads and node postponing and concluded that they no longer considerably improved the efficiency of the code on current hardware. Consequently, we do not include those techniques in the final version of our traversal code.

```
fn iterate_children(t: float,
                    stack: Stack,
                    body: IntersectionFn) -> () {
  // No culling, ignore the entry distance
  let (node, _) = stack.top();

  // Step 1: Load nodes in vector form
  let node_ptr =  node as &[float];
  let tmp0 = ldg(&node_ptr(0) as &simd[float * 4]);
  let tmp1 = ldg(&node_ptr(4) as &simd[float * 4]);
  let tmp2 = ldg(&node_ptr(8) as &simd[float * 4]);
  let child = ldg(&node_ptr(12) as &simd[int * 4]);

  // Step 2: Assemble bounding boxes
  let min1 = vec3(tmp0(0), tmp0(2), tmp1(0));
  let max1 = vec3(tmp0(1), tmp0(3), tmp1(1));

  let min2 = vec3(tmp1(2), tmp2(0), tmp2(2));
  let max2 = vec3(tmp1(3), tmp2(1), tmp2(3));

  // Intersect the two children
  let (tentry1, texit1) = body(min1, max1);
  let (tentry2, texit2) = body(min2, max2);

  let hit1 = tentry1 <= texit1;
  let hit2 = tentry2 <= texit2;
  if !hit1 && !hit2 {
     // No intersection was found
     stack.pop();
  } else {
    // An intersection was found
    if hit1 && hit2 {
      // Both children were intersected:
      // sort the intersections
      if tentry1 < tentry2 {
        let tmp  = child(0);
        child(0) = child(1);
        child(1) = tmp;
      }
      stack.push(child(0), 0.0f);
      stack.set_top(child(1), 0.0f);
    } else {
      // Only one child was intersected
      let next = select(hit1, child(0), child(1));
      stack.set_top(next, 0.0f)
    }
  }
}
```

Listing 5. GPU mapping of the `iterate_children` function.

5 Variants of the Traversal

Our high-level description of the BVH traversal algorithm can be adapted for different purposes. Apart from vectorization width—which can be changed by simply instanciating the traversal code with a different SIMD type—we might want to add other features such as transparency or early ray termination for shadow rays. These features can be added without touching the low-level mappings, as shown in Listing 6.

The `is_terminated` function simply returns true if the ray has hit a triangle. The user provides the `transparency` function which returns a transparency mask for a given point on a triangle. They also provide the `early_exit` variable which controls whether or not the early exit optimization will be applied. Because of partial evaluation (triggered by the `@` sign), the compiler will remove the early exit test if `early_exit == false`.

Since Impala is a functional language, `break` is just a function that can be captured just like any other function. This gives us the opportunity to exit the main traversal loop directly from the deeply nested triangle intersection.

```
while !stack.is_empty() {
  // Capture this loop's exit continuation
  let terminate = break;

  let node = stack.top();

  // Intersect children and update stack
  for min, max in iterate_children(t, stack) {
    intersect_ray_box(org, idir, tmin, t, min, max,
     hit_child);
  }

  // Intersect leaves
  while is_leaf(stack.top()) {
    let leaf = stack.top();

    for id, tri in iterate_triangles(leaf, tris) @{
      let (mut mask, t0, u0, v0) =
        intersect_ray_tri(org, dir, tmin, t, tri);

      mask = transparency(mask, id, u0, v0);

      t = select(mask, t0, t);
      u = select(mask, u0, u);
      v = select(mask, v0, v);
      tri = select(mask, id, tri);

      if early_exit && is_terminated(tri) {
        terminate()
      }
    }

    // Pop node from the stack
    stack.pop();
  }
}
```

Listing 6. Modifications of the high-level traversal loop for early ray termination and transparency. The changes are highlighted in blue.

6 Results

6.1 Benchmarks

We compare our code against hand-tuned, state-of-the-art implementations on both the CPU and a discrete GPU. For all tests, we disabled early ray termination and transparency.

On the CPU, we use the packetized AVX2 traversal routine from Embree 2.4.0 as our reference. This variant is one of the fastest traversal routines made available by Embree, although it may not be the best choice for very incoherent ray distributions. Both Embree's traversal routine and ours use the same acceleration structure (a BVH4 generated by Embree). Even though Impala supports parallelism using TBB, we ran the benchmarks on a single core since Embree 2.4.0 does not provide a multithreaded version of its *kernels*.

Since the Impala compiler framework is built on top of LLVM 3.4.2, we compiled Embree with the same LLVM version using the LLVM-based C++ compiler *clang*. In this setting, Embree is subject to the same low-level optimizations and code generation techniques as RaTrace. For reference, we also give the performance numbers for Embree when compiled with the Intel C/C++ compiler *icc* 16.0.0.

On the GPU, our traversal is benchmarked against a modified version of the Aila's GPU traversal code in which the triangle intersection routine has been replaced. We use the same Möller-Trumbore [26] algorithm as in Embree, instead of a faster intersection routine with precomputed triangle data [44]. All our implementations use this intersection test, which we believe makes comparisons fairer. In this case the acceleration structure for both traversal implementations is an offline-built SBVH [37].

The scenes used for testing all these implementations are presented in Figure 3. We test the pure traversal and intersection routines in different scenarios without performing any shading or other rendering operations:

- **Primary rays** are fired from a pinhole camera and the closest intersection is found.
- **Shadow rays** are shot towards a point light whose position is fixed within the scene and chosen so that the light can reach a good proportion of the overall geometry. The origins of these rays are determined by intersecting the primary rays with the scene.
- **Random rays** are built by taking two random points within the scene bounding box and connecting them with a ray finding the closest intersection.

The results are shown in Table 1. We verified that, for each platform, all corresponding benchmarks operate on the same data structure and the same set of rays.

On the CPU, our traversal routine is always faster than the Embree code compiled with *clang*—which uses LLVM to generate code, as our compiler does—by at least 4%. It is also obvious that the code generated by LLVM, which we have to use, is generally not yet as good as that produced by the Intel compiler. Still, our code performs only slightly slower (within 5%) than Embree compiled with Intel's compiler. On the GPU, our traversal routine performs better than the modified version of Aila's code in every test case, even though the results vary depending on the scene and the ray distribution used.

There are several reasons for this: On the CPU, the control flow and the shape of the loop have a minor influence on the execution time. Performance mainly depends on the number of nodes traversed and the intersection algorithm used. We have consequently designed our abstractions so that the heuristics that decides which children to traverse are almost the same as those used by Embree.

Looking at the machine code generated by both Intel's compiler and Impala/LLVM, we see that our intersection routines contain more AVX register spills; some of them being emitted in critical parts of the loop like the ray-box intersection. Since LLVM is used for register allocation in Impala, it is highly likely that it is responsible for the non-optimal code quality. Using a newer version of LLVM could improve that aspect.

On the GPU, Aila's implementation introduced persistent threads, but this creates some overhead due to the use of a global pool of rays. Since improvements in modern GPU hardware [28] make persistent threads much less effective, our traversal routine performs consistently faster, even for incoherent workloads. Interestingly, we found that adding read-only memory loads gives a speedup (up to 4% in some scenes) at almost no cost in code readability.

(a) Sibenik **(b)** Sponza **(c)** Conference **(d)** San Miguel **(e)** Power Plant

Figure 3. Scenes used for benchmarking different traversal implementations, rendered with ambient occlusion at a resolution of 1024×1024 pixels.

Table 1. Performance of the traversal implementations in **Mrays/s** on an Intel Core i7-4790 CPU with 3.60GHz and 16GB of RAM, and a GeForce GTX 970. The images were rendered at the resolution of 1024×1024. The values are computed from the median of 100 executions in order to discard outliers. The percentages are speed-ups w.r.t. Embree compiled with *icc*, Embree compiled with *clang*, and the work of Aila et al., depending on the platform. The CPU benchmarks are run on a single thread on a single core. The best and worst case of each comparison are highlighted in green and red, respectively.

		CPU			GPU	
Scene	Ray Type	Embree (icc)	Embree (clang)	RaTrace	Aila et al.	RaTrace
Sibenik 75K tris. Figure 3a	Primary	18.17	15.06	17.80 (-2.04%, +18.19%)	336.47	405.01 (+20.37%)
	Shadow	23.93	19.54	23.48 (-1.88%, +20.16%)	459.04	560.44 (+22.09%)
	Random	2.48	2.29	2.39 (-3.63%, +4.37%)	154.83	177.48 (+14.63%)
Sponza 262K tris. Figure 3b	Primary	7.77	6.60	7.46 (-3.99%, +13.03%)	189.45	223.34 (+17.89%)
	Shadow	10.13	8.13	9.82 (-3.06%, +20.79%)	304.17	359.47 (+18.18%)
	Random	2.62	2.41	2.52 (-3.82%, +4.56%)	121.46	141.20 (+16.25%)
Conference 331K tris. Figure 3c	Primary	27.43	23.24	26.80 (-2.30%, +15.32%)	427.96	514.26 (+20.17%)
	Shadow	20.00	16.98	19.86 (-0.70%, +16.96%)	358.66	433.65 (+20.91%)
	Random	5.01	4.61	4.82 (-3.79%, +4.56%)	169.07	181.16 (+7.15%)
San Miguel 7.88M tris. Figure 3d	Primary	4.90	4.31	4.81 (-1.84%, +11.60%)	114.75	132.48 (+15.45%)
	Shadow	4.35	3.90	4.17 (-4.14%, +6.92%)	101.30	122.54 (+20.97%)
	Random	1.52	1.38	1.49 (-1.97%, +7.97%)	90.63	105.27 (+16.15%)
PowerPlant 12.8M tris. Figure 3e	Primary	8.53	7.65	8.43 (-1.17%, +10.20%)	261.13	301.57 (+15.49%)
	Shadow	8.22	7.41	7.77 (-5.47%, +4.86%)	301.02	339.34 (+12.73%)
	Random	4.49	4.22	4.40 (-2.00%, +4.27%)	193.34	242.22 (+25.28%)

6.2 Code Complexity

To demonstrate that our code is simpler and easier to develop, we compare the code complexity of the benchmark implementations. A widely used metric for this purpose is Halstead's complexity metric [18]: It measures the effort required to implement a piece of software. We list three measures in Table 2: Halstead's effort, the estimated number of hours needed to develop each implementation, and the number of lines of code (LoC). The estimated number of hours is computed based on Halstead's measure. We also give the

Table 2. Code complexity of the benchmark implementations. The effort and programming time are based on Halstead's code complexity metric [18]. Halstead's measure gives an estimation of the time taken to implement a piece of software from scratch.

	Effort	Coding time	LoC
Embree	$24.914 \cdot 10^6$	384h	852
RaTrace (CPU)	$2.530 \cdot 10^6$	39h	259
Aila et al.	$2.103 \cdot 10^6$	32h	284
RaTrace (GPU)	$1.986 \cdot 10^6$	31h	274
RaTrace (Common part)	$0.658 \cdot 10^6$	10h	158
RaTrace (CPU mapping)	$0.783 \cdot 10^6$	12h	101
RaTrace (GPU mapping)	$0.473 \cdot 10^6$	7h	116

effort required to implement the common part of the traversal and each mapping *individually*, since this measure is not additive.

For the sake of fairness, we only selected the meaningful and necessary parts of each implementation. In our implementation, we included the complete traversal code with the intersection routines. In the case of Aila's code, we measured the dynamic fetch CUDA kernel, and for Embree, the wrappers for AVX instructions along with the BVH traversal and triangle intersection routines. Removing the wrappers for AVX instructions in Embree would require to adapt the traversal routine to use the intrinsics directly. This would in turn create more complex expressions and hence *increase* the Halstead measure of the code.

The numbers in Table 2 show that the complexity of Embree is significantly higher than that of our CPU implementation. Embree's code is verbose mainly because it relies heavily on AVX intrinsics: There is currently no other way to develop vectorized high-performance software for CPUs in C++. The advantage of our approach is clear: Based on these numbers our implementation is about ten times faster to implement.

According to the same numbers, Aila's implementation has a complexity comparable to ours. Nevertheless, this does not take into account the modularity of the implementation: With our approach it is easy to modify any part of the traversal algorithm independently. Since Aila et al. do not use any abstractions, changing parts of their code without impacting the whole traversal kernel is considerably more challenging.

An important aspect is also that the common part is written only *once* and is reused across *all* different mappings without modification. As we port our code to more hardware architectures (or variants thereof) we avoid rewriting a significant part of it. The same does not hold for either Aila's code or Embree: They do not have the notion of a *common part*. Adding support of different acceleration structures, different triangle layouts, or implementing other modifications would be rather simple in our code base: We just need to modify one of the existing mappings and maybe add some additional abstractions (see Section 5 for an example). The same operations for Embree or Aila's code would be more involved or even require a complete reimplementation.

7 Conclusion and Future Work

We have presented a new approach that allows us to formulate the ray traversal algorithm at a very high level while still achieving excellent performance. Using a language that supports functional programming and a compiler that performs partial evaluation, our approach elegantly separates two major concepts: The *unique* high-level algorithm—that is implemented only once and in a generic way—and the hardware-specific details. Those details are provided separately for each hardware architecture and are typically implemented by a hardware expert.

Any "overhead" incurred by these *conceptual* abstractions and language constructs is eliminated with the use of a recently introduced compiler technology. Our results show that our code is much easier to write *and at the same time* as fast and efficient as current state-of-the-art, heavily hand-optimized implementations.

As future work we would like to apply the same techniques to other parts of a renderer: They could be abstracted as well and run efficiently on both CPUs and GPUs. The abstraction of the shading system will be particularly interesting since it requires just-in-time compilation. Also, we could use partial evaluation to specialize the ray traversal kernels presented here: We would then optimize our code for a specific use-case such as coherent or incoherent ray tracing with little to no overhead.

The full implementation of the traversal and its different mappings are distributed under the LGPL v3 and are available at https://github.com/AnyDSL/traversal.

Acknowledgments

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects as well as by the Intel Visual Computing Institute Saarbrücken. It is also co-funded by the European Union (EU), as part of the Dreamspace project.

References

[1] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High-Performance Graphics (HPG)*. ACM, 145–149. https://doi.org/10.1145/1572769.1572792

[2] Timo Aila, Samuli Laine, and Tero Karras. 2012. *Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum.* Technical Report NVR-2012-002. NVIDIA Technical Report.

[3] L.O Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. Københavns Universitet. Datalogisk Institut.

[4] P.H. Andersen. 1995. *Partial Evaluation Applied to Ray Tracing.* DIKU Research Report 95/2.

[5] Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic Ray Stream Traversal. *ACM Trans. Graph.* 33, 4, Article 151 (2014), 9 pages. https://doi.org/10.1145/2601097.2601222

[6] Carsten Benthin and Ingo Wald. 2009. Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. In *High-Performance Graphics*. https://doi.org/10.1145/1572769.1572791

[7] Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark. 2012. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics* 18, 9 (2012), 1438–1448. https://doi.org/10.1109/TVCG.2011.277

[8] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 89–100. https://doi.org/10.1109/PACT.2011.15

[9] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. 2010. Language virtualization for heterogeneous parallel computing. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 835–847. https://doi.org/10.1145/1869459.1869527

[11] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 111–120. https://doi.org/10.1145/2254064.2254079

[12] Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed Ray Tracing. *SIGGRAPH Comput. Graph.* 18, 3 (1984), 137–145. https://doi.org/10.1145/964965.808590

[13] Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*. Eurographics Association, 1225–1233. https://doi.org/10.1111/j.1467-8659.2008.01261.x

[14] Zach DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 9:1–9:12. https://doi.org/10.1145/2063384.2063396

[15] Tim Foley and Jeremy Sugerman. 2005. KD-tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. ACM, 15–22. https://doi.org/10.1145/1071866.1071869

[16] Iliyan Georgiev and Philipp Slusallek. 2008. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE Symposium on Interactive Ray Tracing (RT)*. 115–122. https://doi.org/10.1109/RT.2008.4634631

[17] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. 113–118. https://doi.org/10.1109/RT.2007.4342598

[18] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

[19] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE)*. 137–148. https://doi.org/10.1145/1449913.1449935

[20] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. 2007. Interactive K-d Tree GPU Raytracing. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM, 167–174. https://doi.org/10.1145/1230100.1230129

[21] P. Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse (ICSR)*. IEEE Computer Society, 134–. http://dl.acm.org/citation.cfm?id=551789.853532

[22] Timothy L. Kay and James T. Kajiya. 1986. Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.* 20, 4 (1986), 269–278. https://doi.org/10.1145/15886.15916

[23] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015. Shallow Embedding of DSLs via Online Partial Evaluation. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 11–20. https://doi.org/10.1145/2814204.2814208

[24] Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A Graph-Based Higher-Order Intermediate Representation. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 202–212. https://doi.org/10.1109/CGO.2015.7054200

[25] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. HIPA^{cc}: A Domain-Specific Language and Compiler for Image Processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (2016), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

[26] Tomas Möller and Ben Trumbore. 1997. Fast, Minimum Storage Ray-Triangle Intersection. *J. Graphics, GPU, & Game Tools* 2, 1 (1997). https://doi.org/10.1080/10867651.1997.10487468

[27] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael D. McCool, Anwar M. Ghuloum, Stefanus Du Toit, Zhi-Gang Wang, Zhaohui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. 2011. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*. 224–235. https://doi.org/10.1109/CGO.2011.5764690

[28] NVIDIA. 2014. *Whitepaper: NVIDIA GeForce GTX 980.* Technical Report. NVIDIA Corporation.

[29] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 125–134. https://doi.org/10.1145/2517208.2517228

[30] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. 1999. Interactive Ray Tracing. In *Proceedings of the Symposium on Interactive 3D Graphics*. ACM, 119–126. https://doi.org/10.1145/300523.300537

[31] Steven G. 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* (2010). https://doi.org/10.1145/1778765.1778803

[32] M. Pharr and W. R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *In Proceedings of Innovative Parallel Computing (InPar)*. https://doi.org/10.1109/InPar.2012.6339601

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 519–530. https://doi.org/10.1145/2462156.2462176

[34] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level Ray Tracing Algorithm. *ACM Trans. Graph.* 24, 3 (2005), 1176–1185. https://doi.org/10.1145/1073204.1073329

[35] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE)*. 127–136. https://doi.org/10.1145/1868294.1868314

[36] Kai Selgrad, Alexander Lier, Franz Köferl, Marc Stamminger, and Daniel Lohmann. 2015. Lightweight, Generative Variant Exploration for High-performance Graphics Applications. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 141–150. https://doi.org/10.1145/2814204.2814220

[37] Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of the Conference on High-Performance Graphics (HPG)*. ACM, 7–13. https://doi.org/10.1145/1572769.1572771

[38] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*. 609–616.

[39] John A. Tsakok. 2009. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *Proceedings of the Conference on High-Performance Graphics (HPG)*. ACM, 151–158. https://doi.org/10.1145/1572769.1572793

[40] Ingo Wald. 2005. The OpenRT-API. In *ACM SIGGRAPH Courses*. ACM, Article 21. https://doi.org/10.1145/1198555.1198760

[41] Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets: Efficient SIMD single-ray traversal using multibranching BVHs. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*. 49–57. https://doi.org/10.1109/RT.2008.4634620

[42] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum* (2001). https://doi.org/10.1111/1467-8659.00508

[43] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4, Article 143 (2014), 8 pages. https://doi.org/10.1145/2601097.2601199

[44] Sven Woop. 2004. *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Technical Report. Saarland University.