Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering

WENZEL JAKOB, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
SÉBASTIEN SPEIERER, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
NICOLAS ROUSSEL, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
DELIO VICINI, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

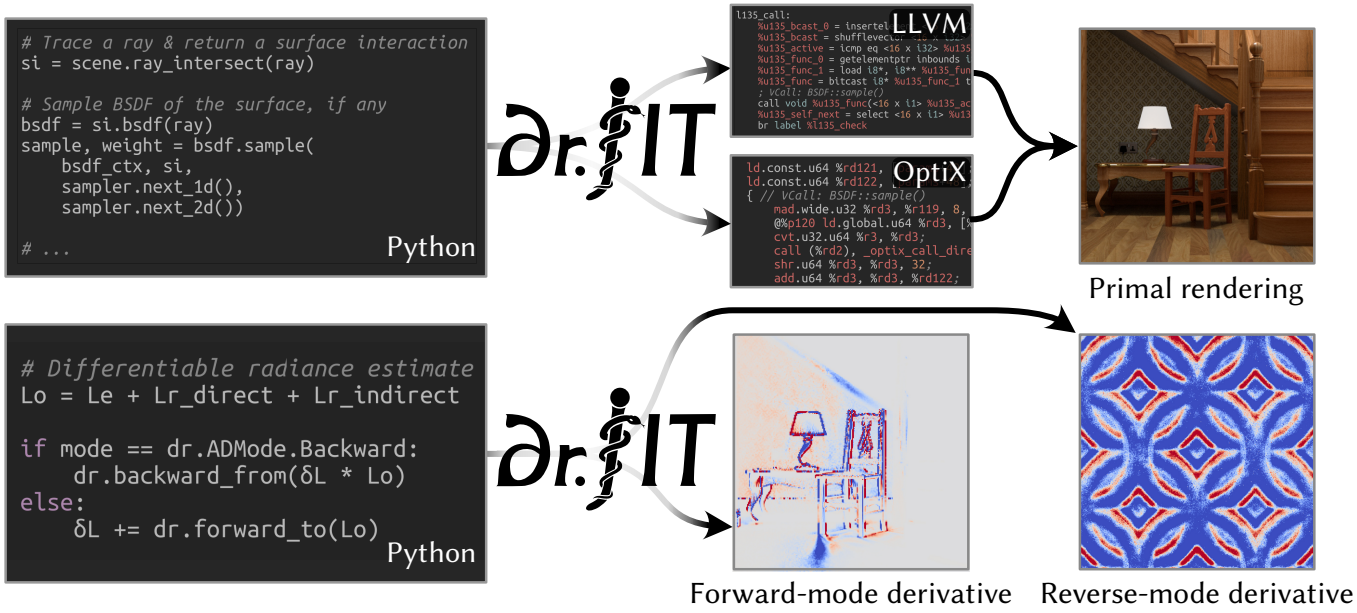


Fig. 1. Dr.Jit is a domain-specific compiler for physically-based (differentiable) rendering. When Dr.Jit executes a rendering algorithm, it generates a *trace*: a large graph comprised of arithmetic, loops, ray tracing operations, and polymorphic calls that exchange information between the rendering algorithm and scene objects (shapes, BSDFs, textures, emitters, etc.). Dr.Jit specializes this graph to the provided scene and compiles it into a large data-parallel kernel ("megakernel") via LLVM or OptiX backends, achieving geometric mean GPU speedups of 3.70× (vs. Mitsuba 2) and 2.14× (vs. PBRT 4). While helpful for ordinary rendering, the main purpose of Dr.Jit is to dynamically compile *differential simulations*. Recent methods in this area decompose a larger differentiation task into a series of incremental steps, which requires an unusually fine-grained approach to automatic differentiation. Dr.Jit supports such transformations in forward and reverse modes: the former computes a perturbation in image space, which is helpful for debugging and visualization. The latter provides derivatives in parameter space (e.g. texels of the wallpaper) for simultaneous optimization of large numbers of unknowns.

Dr.Jit is a new just-in-time compiler for physically based rendering and its derivative. Dr.Jit expedites research on these topics in two ways: first, it traces high-level simulation code (e.g., written in Python) and aggressively simplifies and specializes the resulting program representation, producing data-parallel kernels with state-of-the-art performance on CPUs and GPUs.

Second, it simplifies the development of differentiable rendering algorithms. Efficient methods in this area turn the derivative of a simulation into a simulation of the derivative. Dr.Jit provides fine-grained control over the process of automatic differentiation to help with this transformation.

Specialization is particularly helpful in the context of differentiation, since large parts of the simulation ultimately do not influence the computed gradients. Dr.Jit tracks data dependencies globally to find and remove redundant computation.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; • **Mathematics of computing** → **Automatic differentiation**; • **Computing methodologies** → **Rendering**.

Additional Key Words and Phrases: differentiable rendering, just-in-time compilation, automatic differentiation, megakernel, GPU rendering

ACM Reference Format:
Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *ACM Trans. Graph.* 41, 4, Article 124 (July 2022), 19 pages. https://doi.org/10.1145/3528223.3530099

Authors' addresses: Wenzel Jakob, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, wenzel.jakob@epfl.ch; Sébastien Speierer, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, sebastien.speierer@epfl.ch; Nicolas Roussel, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, nicolas.roussel@epfl.ch; Delio Vicini, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, delio.vicini@epfl.ch.

0730-0301/2022/7-ART124 $15.00
https://doi.org/10.1145/3528223.3530099

1 INTRODUCTION

Recent progress in the area of *physically based differentiable rendering* (henceforth "PBDR") has led to the development of methods that can differentiate light transport simulations with respect to arbitrary scene parameters. Combined with a gradient-based optimizer, they can solve nonlinear problems involving large sets of unknowns. Diverse scientific and engineering disciplines require the inverse analysis of images and stand to benefit from these developments.

While the theory of PBDR continues to evolve, practical aspects have remained a persistent challenge. For example, the reverse-mode derivative of a conceptually simple algorithm like path tracing [Kajiya 1986] with precautions for linear time complexity [Vicini et al. 2021] and unbiased visibility handling [Bangaru et al. 2020] turns into an enormously complicated function. Correct implementation of such a large and intricate program is near-impossible even for experts in the field. Mere correctness is also unsatisfactory: optimizations tend to run for thousands of iterations, hence the resulting program needs to be fast. It is evident that better tools are need bridge this conspicuous gap between PBDR theory and practice.

The design of Dr.Jit was guided by a single unifying objective: it should provide a practical and efficient foundation for work in this area. Most architectural decisions are direct consequences of this overarching goal. For example, consider the differentiation step that is implicit in differentiable rendering. Manual differentiation is tedious and error-prone, hence it is logical that the system should build on *automatic differentiation* (AD) to simplify development.

However, the needs of PBDR are more specific: standard use of AD to differentiate a rendering algorithm produces an inefficient and biased derivative that precludes many applications. Recent work addresses inefficiencies using physical reciprocity [Nimier-David et al. 2020] and arithmetic invertibility [Vicini et al. 2021] to turn the derivative of a simulation into a simulation of the derivative, while re-parameterizing the integration domain to remove bias [Loubet et al. 2019]. These steps move the differentiation operation into the random walk, where it introduces partial derivative terms at each scattering event. This has implications on the design of the system: the derivatives must somehow be (pre-)compiled, since the machinery of AD is too slow to be used dynamically at such high rates.

Differentiating a simulation changes the underlying computation, but the details of this change depend on the scene, simulation algorithm, and optimization task. When the optimization only targets a subset of the scene's parameters, it is desirable that the system uses this information to remove steps that cannot influence the computed gradient. The dynamic nature of this problem calls for a similarly dynamic approach to compilation, which is why we pursue an approach centered around *just-in-time* (JIT) compilation.

Effective use of modern computing hardware requires that the program is organized into data-parallel phases known as *kernels*. Several kernels are generally needed to handle data dependencies, which must then exchange information through device memory. This inter-kernel communication comes at a cost in terms of storage and memory bandwidth, hence the specific manner in which a computation is partitioned into kernels can have a pronounced impact on performance. In the case of PBDR, the simulation parallelizes over millions of Monte Carlo samples that represent a large amount of program state. In our experiments, we find that it is almost always preferable that the Monte Carlo integration occurs within a *megakernel*, i.e., a large kernel containing all program instructions needed to evaluate the integrand. Most sample state can then be stored in registers, reducing memory usage and inter-kernel communication.

Finally, physically-based rendering algorithms are commonly expressed using *subtype polymorphism* to dynamically dispatch method calls from abstract component interfaces (e.g., a material encountered by a ray) to concrete implementations (e.g., a woven fabric or a rough metallic surface). The ability to represent, differentiate, and optimize such polymorphic constructions benefits performance.

Taking stock, we arrive at the following set of requirements:

1. The system must dynamically generate specialized code for a given scene, rendering algorithm, and optimization task.
2. Compilation should be able to ensure that any use of Monte Carlo integration remains fully contained within a megakernel.
3. The system must scale. Challenges include fine-grained AD, thousands of volume scattering events, large numbers of shapes and materials accessed through polymorphic abstractions.

Dr.Jit addresses these requirements using an approach based on *tracing*. It executes simulation code in a deferred manner by recording encountered operations into a *trace* for subsequent compilation and execution on a target device. This process is automatic and language-agnostic (Python and C++ are currently supported).

A key difference to Mitsuba 2 [Nimier-David et al. 2019], which also uses tracing, is that Dr.Jit must handle a larger set of operations to guarantee successful megakernel generation. A Dr.Jit (differential) rendering step captures loops, polymorphism, and ray intersection operations without interruption, returning an unevaluated image in the form of a *very large* trace that encompasses the rendering algorithm and implementations of all referenced scene objects including materials, textures, volumes, light sources, etc.

This global representation reveals optimization opportunities. For example, suppose that rendering does not integrate over time: it is then safe to remove time-related variables from loops and function interfaces. Similarly, the derivatives of many program variables do not influence the computed parameter gradient and can be deleted.

Dr.Jit finally compiles and evaluates the trace via OptiX [Parker et al. 2010] or LLVM [Lattner and Adve 2004]. The former produces GPU kernels leveraging ray tracing hardware acceleration, while the latter generates vectorized code for diverse CPU architectures.

In addition to Dr.Jit, we also present Mitsuba 3, a new version of the Mitsuba renderer that builds on Dr.Jit. Both projects are available under an open source license at https://github.com/mitsuba-renderer/drjit and https://github.com/mitsuba-renderer/mitsuba3.

Differentiable rendering subsumes ordinary rendering, and Dr.Jit achieves state-of-the-art performance on both tasks. We substantiate all performance-related observations experimentally. Following a review of related work, the remainder of this article discusses compilation (Section 3) and differentiation (Section 4) in turn.

2 RELATED WORK AND BACKGROUND

2.1 Array programming

Increasing interest in machine learning in recent years has precipitated the creation of numerous frameworks that combine AD with

n-dimensional array representations. They use JIT backends like XLA [Google 2017] to fuse operations into efficient kernels. Given widespread success and obvious similarities to Dr.Jit, it is not unreasonable to wonder whether this article could have been cut short by a recommendation to implement PBDR methods on top of such a framework? We investigate this question in Section 3 and find that PBDR workloads exhibit characteristics that are unusual in the array programming setting, causing them to fall off the fast path.

2.2 Automatic Differentiation

Manual differentiation of mathematical expressions is error-prone and mechanical. The natural desire to delegate this task to a computer began with pioneering work in the 1950s and 1970s [Nolan 1953; Wengert 1964; Linnainmaa 1976] followed by comprehensive study in the 1980s [Speelpenning 1980; Griewank et al. 1989]. Griewank and Walther's book [2008] reviews what has been learned about AD over the course of these many decades.

Given the consolidated understanding of the mathematical structure and asymptotic complexity of various derivative propagation strategies, there is a surprising degree of variety when it comes to how AD should be exposed to the user. The space of methods includes tracing, source-to-source transformation of abstract syntax trees or *intermediate representations* (IR), and hybrids combining tracing with transformation. Differentiation can target scalar, dense, or sparse array programs in forward, reverse-, and mixed modes, computing gradients or higher-order derivatives of pure and impure functions. Covering all techniques in detail is far beyond the scope of this paper, and we refer to a recent survey by Baidin et al. [2018]. This section only covers core concepts, noteworthy related methods for first-order derivatives, and their relationship to our approach.

Directionality. The main high-level flavors of AD are the *forward* and *reverse* (also known as *adjoint* or *backward*) modes. Forward mode evaluates a *Jacobian-vector-product* (often abbreviated "JVP") of the form $\boldsymbol{\delta y} = \mathbf{J}_f \boldsymbol{\delta x}$, where $\mathbf{J}_f$ is the Jacobian of the *primal* (i.e., original) computation $\mathbf{y} = f(\mathbf{x})$, and reverse mode evaluates a *vector-Jacobian-product* ("VJP") of the form $\boldsymbol{\delta x} = \boldsymbol{\delta y}^T \mathbf{J}_f$. Both can in principle compute the same derivatives, but forward mode does this more efficiently when the function being differentiated has few inputs (ideally just one), while reverse mode is efficient if it has few outputs. Realistic scene descriptions have million of unknowns, hence practical differentiable rendering depends on reverse mode.

Reverse mode. The key issue with reverse mode is that it inverts the data dependencies of the original program. The derivative of this reversed program references intermediate steps of the primal calculation, which raises the age-old question of how they should be obtained. Exhaustive storage is simple but does not scale, as modern processors can generate many terabytes of intermediate state per second. The usual remedy is to only store this state at a sparse set of *checkpoints* with later recovery via reevaluation from the nearest one [Volin and Ostrovskii 1985]. Automatic recursive usage of this pattern reduces storage and runtime overheads of a program with t operations to a factor of $O(\log t)$ [Siskind and Pearlmutter 2018].

Adjoints. When available, *custom adjoints* are generally preferable to checkpointing. These differentiation techniques exploit problem-specific traits to reduce storage and reevaluation overheads. For example, differentiating through all steps of a multivariate Newton's method is unnecessarily inefficient, since the implicit function theorem provides the answer directly from the iteration's fixed point. Similarly, the solution of an ordinary differential equation admits an efficient adjoint that reverses time [Pontryagin 1962].

Adjoints for rendering. Light transport also admits custom adjoints: *Radiative Backpropagation* (RB) [Nimier-David et al. 2020] and *Path Replay Backpropagation* (PRB) [Vicini et al. 2021] transform the derivative of a simulation into an equivalent and more efficient simulation of *derivative radiation*. Section 4.1 discusses them further.

Tracing. Assuming that efficient adjoints are available, the next important question is how the computation to be differentiated should be ingested. Methods based on *tracing* record an evaluation trace (also referred to as a *Wengert tape* or *computation graph*) of all differentiable arithmetic operations; differentiation traverses this trace once more to propagate derivatives in forward or reverse mode. The trace ignores control flow constructs and stores unrolled versions of all loops and taken branches. Tracing has seen widespread adoption through tools like PyTorch [Paszke et al. 2017], in which a typical neural network produces a trace with a few hundred high-level operations. The reverse-mode sweep then invokes efficient adjoints of each operation. Tracing inherits the usual caveats of reverse mode—for example, differentiating long-running loops can be challenging. It also adds runtime overheads that can normally be amortized when differentiating array programs, but they dominate in *scalar programs* that manipulate individual floating point values. In iterative computations, tracing is often performed repeatedly, which causes further overheads and inhibits optimization.

Source transformation. AD via *source transformation* converts the source code of a program into its derivative. In essence, differentiation becomes a one-time compilation step, which enables practical differentiation of scalar programs. Figure 2 illustrates this on a (not particularly good) implementation of an integer power x^n, with JVP/VJP variants generated by Tapenade [Hascoet and Pascual 2013]. A key advantage of this approach is the preservation of control flow: unlike an unrolled trace, the programs in Figure 2 work regardless of the value of n (the number of loop iterations).

```
def pow(x: float, n: int):
    y = 1
    for i in range(n):
        y *= x
    return y
```

```
def pow_jvp(x, n, dx):
    y, dy = 1, 0
    for i in range(n):
        dy = x*dy + y*dx
        y *= x
    return dy
```

```
def pow_vjp(x, n, dy):
    y, dx = 1, 0
    stack = []
    for i in range(n):
        stack.append(y)
        y *= x
    for i in reversed(range(n)):
        y = stack.pop()
        dx += y*dy
        dy *= x
    return dx
```

Fig. 2. Derivatives of a program obtained using *source transformation*. Here, pow raises x to the n-th power ($n \in \mathbb{N}$). The JVP/VJP versions were generated by Tapenade [Hascoet and Pascual 2013] and translated to Python. Both require the primal function arguments as input. The JVP further takes the function input derivatives and converts them into output derivatives, while the VJP takes output derivatives and converts them into input derivatives.

Loops continue to be a nuisance, however: observe how the VJP requires two of them: the first records all intermediate loop state into a `stack` variable that is later consumed by a reversed loop. This poses difficulties on massively parallel architecture like GPUs: stack or *shadow memory* must be provisioned for all threads running in parallel, while handling worst-case requirements that are generally unknown. The JVP and VJP are equivalent in this specific example, since `pow` only has a single differentiable argument and return value.

Curiously, optimizations that are straightforward in one AD approach can become relatively difficult in another. For example, only a subset of program variables usually affects the final gradient; it is desirable that the AD system recognizes this to avoid unnecessary adjoint evaluations. In tracing AD, this optimization happens automatically, since traversal along data dependencies cannot reach irrelevant variables. To achieve the same goal, source transformation tools must perform a more involved *activity analysis* [Bischof et al. 1992], which is a data-flow analysis that conservatively propagates derivative liveness through the control flow graph until this process reaches a fixed point.

Four notable source transformation tools are Tapenade [Hascoet and Pascual 2013], Stalin∇ [Pearlmutter and Siskind 2008], Zygote [Innes 2019], and Enzyme [Moses and Churavy 2020; Moses et al. 2021]. Tapenade uses sophisticated data-flow analyses to generate optimized derivatives of scalar C or Fortran programs involving pointers and array mutation. Stalin∇ operates on λ-calculus IR and performs source-to-source transformations using runtime reflection. The *callee-derives* approach in their work shares some of the motivation of Dr.Jit's specialization of polymorphic derivatives. Zygote operates on typed IR of the Julia language and composes higher-level adjoints, while Enzyme differentiates code following lowering to LLVM IR. As with optimizing compilers, the IR's abstraction level can facilitate or exacerbate certain tasks, hence the suitability of these tools depends nature of the problem to be differentiated.

Hybrid systems. Tracing and source transformation are merely the extreme points of a large space of *hybrid* techniques with interesting trade-offs. For example, PyTorch [2017] programs can be traced into a domain-specific language for subsequent compilation, which removes tracing overheads and enables optimizations. JAX [Bradbury et al. 2018] traces functions into an expressive IR that can be differentiated or parallelized. Compilation proceeds via lowering into XLA [Google 2017] that performs further tensorial optimizations. Dr.Jit is also a hybrid in this classification: it ingests computation using tracing but generates kernels that preserve the control flow (loops, subroutines) of the original program. Its output thus resembles that of a source transformation–based tool.

Functional languages. Following Elliot's [2018] formalization of backpropagation using category theory and continuation passing, multiple works have proposed languages that combine the expressiveness of higher-order functional programming with the efficiency of imperative languages. This includes the previously mentioned Stalin∇ [Pearlmutter and Siskind 2008] and languages like dF̃ [Shaikhha et al. 2019] and Dex [Paszke et al. 2021]. Dr.Jit adopts an imperative approach to interoperate with C++ and Python. That said, the authors find combinations of PBDR with differentiable functional programming promising and worthy of future exploration.

2.3 Graphics and compilers

Graphics applications have a near-insatiable thirst for floating point operations, usually operating at the limits of what is possible on present hardware. Their needs are often not well-served by existing programming languages and compilers, which has motivated specialized systems with broad impact. For example, the *RSL* shading language [Hanrahan and Lawson 1990] pioneered the use of programmability in a previously mostly static graphics stack, using JIT-compilation and scene-specific specialization analogous to similar steps in Dr.Jit. Following increased programmability of GPUs through shading languages like *Cg* [Mark et al. 2003], the *Brook* [Buck et al. 2004] project was instrumental in repurposing GPUs for general-purpose computation ("GPGPU").

SMASH [McCool et al. 2002] introduced the idea of metaprogramming shaders by tracing arithmetic in a host language, and *Sh* [McCool et al. 2004] further specialized generated code. Spark [Foley and Hanrahan 2011] raised the level of abstraction by disentangling features from the underlying GPU pipeline stages, and Slang [He et al. 2018] modularized shader development by separating aspects like lighting, materials, and camera transformations. The system parses high-level C++-style code and specializes the resulting IR before emitting low-level shader code for various platforms.

Halide [Ragan-Kelley et al. 2012] simplifies the design of image processing pipelines by separating computation and *schedule*, which encompasses placement, parallelization, vectorization, and blocking. A differentiable extension [Li et al. 2018b] enables end-to-end optimization of image processing pipelines. The predictable structure of the underlying stencils enables highly effective optimizations like scatter-gather conversion during differentiation. *Taichi* [Hu et al. 2019] is a general-purpose parallel programming language with an emphasis on dynamical simulation and sparse data structures. *DiffTaichi* [Hu et al. 2020] endows *Taichi* with an efficient differentiation operator that preserves the megakernel structure of the simulation code. While *DiffTaichi*, *Halide*, and Dr.Jit each target different types of programs, they share a common focus on optimizing memory access patterns when a computation is differentiated.

A number of PBR systems rely on specialized compilation techniques: *MoonRay* [Lee et al. 2017] traces vectorized wavefronts, and *Manuka* [Fascione et al. 2018] JIT-compiles shader graphs for a vectorized parallel batch shading phase. Building on the *AnyDSL* [Leißa et al. 2018] partial evaluation framework, *Rodent* [Pérard-Gayot et al. 2019] compiles specialized renders for each scene, sharing some of the motivation of Dr.Jit. *Mitsuba 2* [Nimier-David et al. 2019] builds on the *Enoki* [Jakob 2019] library to retarget generic specifications of rendering algorithms and scene objects to diverse applications including differentiation, vectorization, spectral rendering, and polarization. Dr.Jit was designed as a replacement for Enoki, enabling experimentation with compilation and differentiation using the infrastructure of an existing renderer.

Given the favorable discussion of megakernels in Section 1 we should also review their disadvantages, which were studied by Laine et al. [2013]: megakernels tend to contain large amounts of code connected via branch instructions. Branch divergence can then reduce the effectiveness of vectorized program execution, while high register usage interferes with the latency-hiding mechanism of GPUs. Dr.Jit can be used to explore the impact of kernel size

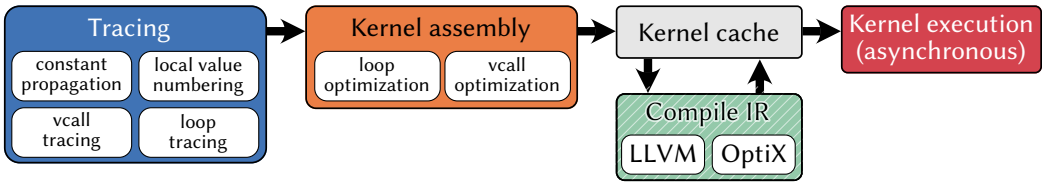


Fig. 3. The five main phases of Dr.Jit. *Tracing* executes a Python or C++ program using custom arithmetic types that record operations into a graph data structure. Basic optimizations remove redundancies and reduce the size of the program. Tracing of loops and polymorphic constructs (*vcalls*, i.e., virtual method calls) requires special precautions at this stage. *Kernel assembly* removes redundancies at a global level and produces a program in the desired intermediate representation (LLVM IR or PTX). Tracing and kernel assembly are highly optimized (on the order of 1-15ms in typical cases). A subsequent backend compilation step converts the generated IR into executable machine code. Backend compilation is relatively costly, hence Dr.Jit consults an in-memory and on-disk kernel cache to see if this computation was previously encountered. Caching is a good fit for the repetitive computation performed by gradient-based optimizers.

on performance: besides producing megakernels, it is also able to emit different granularities of *wavefronts*, which refers to a sequence of smaller kernels with intense data exchange through device memory. Laine et al. [2013] compared both extremes and found wavefront execution to be the superior execution model. However, nearly a decade and four hardware generations later, our experiments indicate that the overall balance seems to have shifted back towards megakernels, at least for the types of workloads investigated in this article. We expect that there will still be a point where a megakernel is simply *too large* to run reasonably on a GPU.

3 JUST-IN-TIME COMPILATION

We now turn to Dr.Jit's computational substrate and postpone most discussion of differentiation to Section 4. Any use of the term *tracing* in this section thus refers to capturing computation for later compilation and is unrelated to AD. The term is also not to be confused with *ray tracing*, which refers to a specific geometric intersection operation that can appear within a trace.

Figure 3 illustrates the high-level pipeline: following tracing, kernel assembly generates IR requiring backend compilation into machine code. In typical PBDR usage, this is only necessary during the first gradient descent step. The final pipeline stage launches the kernel for asynchronous execution on the CPU (via a thread pool) or the GPU (via *CUDA*/*OptiX*) so that tracing and kernel execution can continue in parallel. *OptiX* [Parker et al. 2010] is a domain-specific compiler that accelerates ray tracing on NVIDIA GPUs, and which offloads these operations onto dedicated hardware cores if available.

3.1 Running example

We will now walk through the implementation of a simple *ambient occlusion* integrator and use this as an opportunity to introduce major system components, starting with low-level details and then progressively zooming out. Our program begins by importing[1] Dr.Jit along with a floating point and an integer type.

```
import drjit as dr
from drjit.cuda import Float, UInt32
```

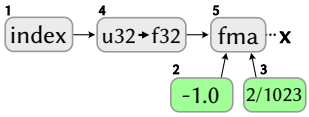
While types like `Float` and `UInt32` are suggestive of an internal scalar representation, they represent dynamically sized 1D arrays. Instances of them are best thought of as a scalar variable declaration within a loop of the form "`for index in range(..)`", where `index` will usually refer to the Monte Carlo sample being computed.

Operations involving these capitalized types become part of the trace, while builtin Python types (`float`, `int`) and control flow statements are invisible to Dr.Jit. In other words, this is a *metaprogram*, whose execution determines what computation will eventually take place on the target device.

The next line creates a `Float` variable containing 1024 evenly spaced numbers covering the interval $[-1, 1]$ that we will shortly use to generate primary camera rays.

```
x = dr.linspace(Float, -1, 1, num=1024)
```

The trace of this expression consists of five variables: the aforementioned loop `index` variable, an int-to-float cast, and a *fused multiply-add* (FMA) referencing two *literal constants* shown in green:



A variable trace usually contains at least one use of the `index` variable, otherwise it is uniform. Each temporary/variable is identified by a number (`x`, e.g., points to #5). An associative data structure maps this index to a record describing the operation and its dependencies. The next step of our example creates another 1D array and invokes `dr.meshgrid` to expand both arrays into 2D grid coordinates.

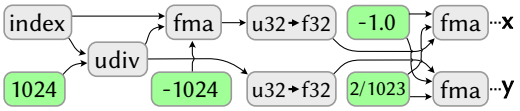
```
y = dr.linspace(Float, -1, 1, num=1024)
x, y = dr.meshgrid(x, y)
```

Several points are worthy of note: the computation of the `y` variable is of course redundant. Dr.Jit detects this using *local value numbering*, an optimization that uses the variable details (operation type, input dependencies) as *key* to query an auxiliary *inverse* version of the associative mapping. If an equivalent variable exists, it will be reused. The system also performs constant folding/propagation and basic algebraic simplifications (e.g. `fma(a, b, 0) = a*b`; `a*1=a`) while tracing. These optimizations are not important in our example, but they are effective in combination with AD that tends to generate many operations of this type. Both steps are cheap to do while tracing, and they reduce the size of the IR passed to OptiX/LLVM.

The function `dr.meshgrid()` computes row and column indices and then calls `dr.gather()` to read from the input arrays `x` and `y`.

[1]The shown source code fragments use the Python language, but the system is also usable via a near-identical C++ API. JIT-compilation and differentiation usually trace combinations of code written in both languages. Operations like `linspace` and `meshgrid` imitate their eponymous counterparts in other array programming tools.

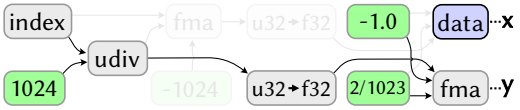
Both represent computation that has not occurred yet, and `dr.gather` is thus able to perform the indexing operation symbolically by cloning their graph representation and rewriting the `index` variable. Along with further application of the previously mentioned optimizations (value numbering, constant propagation), this produces the following combined trace:



Suppose that the example takes place in an interactive session, and the user wishes to check the contents of the `x` variable at this point.

```
>>> print(x)
[-1.0, -0.998, -0.996, .. 1048570 skipped .., 0.996, 0.998, 1.0]
```

Accessing array contents triggers evaluation via `dr.eval(..)`, which compiles and launches a fused kernel that commits the requested variables(s) to device memory. Further use of `x` references the stored version, hence parts of the trace that are no longer needed expire.



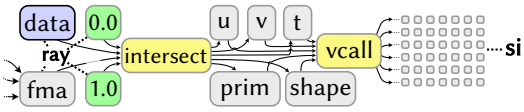
We are now almost ready to process a set of primary rays, using the 2D grid as image-space offsets in an orthographic camera model:

```
from mitsuba import Ray3f, Point3f, Vector3f, load_file
ray = Ray3f(o=Point3f(x, y, 0), d=Vector3f(0, 0, 1))
scene = load_file('scene.xml')
si = scene.ray_intersect(ray)
```

Mitsuba 3 uses Dr.Jit-provided types throughout all interfaces; fixed-size arrays like `Point3f` simply wrap several JIT variables that become part of the generated program. The ray tracing operation requires further elaboration: its implementation looks as follows:

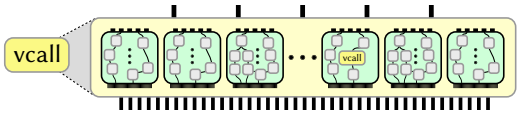
```
class Scene:  # .. (most definitions omitted) ..
    def ray_intersect(self, ray: Ray3f) -> SurfaceInteraction3f:
        pi = self.ray_intersect_preliminary(ray)
        return pi.shape.compute_surface_interaction(pi)
```

It first finds a *preliminary intersection* comprised of the intersected shape, primitive ID, ray depth and UV coordinates (our example is simplified here to hide complexities like instancing). The second step invokes a method on the intersected shape (`pi.shape`) so that it can refine this preliminary intersection into a detailed `SurfaceInteraction3f` type describing the differential geometry at the intersection point. Within Mitsuba 3, this refined intersection maps to a large number of variables (41-45 depending on the variant of the renderer). Following these last steps, the trace looks as follows (repeated parts on the left are omitted):



The yellow boxes represent complex operations with custom code generation hooks. During kernel assembly, `intersect` produces IR that invokes a ray tracing accelerator (Embree [Wald et al. 2014] on the CPU and OptiX [Parker et al. 2010] on the GPU).

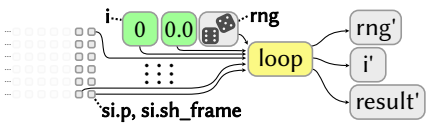
The `pi.shape` member of the preliminary intersection refers to an (as of yet unevaluated) array of 1024×1024 pointers to arbitrary shape implementations. The yellow operation labeled "`vcall`" represents the dynamic dispatch step (*virtual function call* in C++ terminology) needed to resolve the polymorphic nature of this operation. Dr.Jit does so by tracing *all reachable* method implementations (scene objects register themselves with Dr.Jit to enable this). The operation can thus be interpreted as a large-scale demultiplexer-multiplexer that routes arguments and return values to/from instances. During kernel assembly, it produces IR performing an indirect branch to one of multiple subroutines (in other words, the indirection is preserved in the generated program).



Several inefficiencies may be apparent to the reader at this point; we will soon explain how they are resolved. The final part of our example traces 128 ambient occlusion rays starting at the intersected position. For this, we require a random number generator and a looping mechanism. While a standard Python **`for`** loop works, it would unroll the loop body 128 times and produce an unnecessarily large trace. As before, it is desirable that tracing preserves the structure of the input program. We instead instantiate a Dr.Jit `Loop` object along with a pseudorandom number generator (`PCG32`), a counter (`i`), and a variable representing the final result.

```
from drjit.cuda import Loop, PCG32
rng = PCG32(size=1024 * 1024)
i, result = UInt32(0), Float(0)
loop = Loop(state=lambda: (rng, i, result))
while loop(si.is_valid() & (i < 128)):
    # ... loop body ...
    i += 1
```

This loop runs for only *one* iteration (the condition `loop(..)` returns **`False`** in the second round). This suffices to capture the effects of an individual loop iteration, which is all that is needed to wire[2] it into the generated kernel. A downside of our approach for embedding tracing into a host language is that Dr.Jit must know about the loop's *state variables*, which refers to variables that are modified in the body and either accessed in subsequent iteration or following termination. They are specified using a lambda function, which our implementation invokes before and after the loop to catch redefinitions of Python variables. The trace now looks as follows (`rng` seeding omitted for simplicity).



The specific loop of our example samples the cosine-weighted hemisphere and traces shadow rays.

[2]This involves the insertion of basic block boundaries and `Phi` nodes into the generated IR, which is in static single assignment (SSA) form.

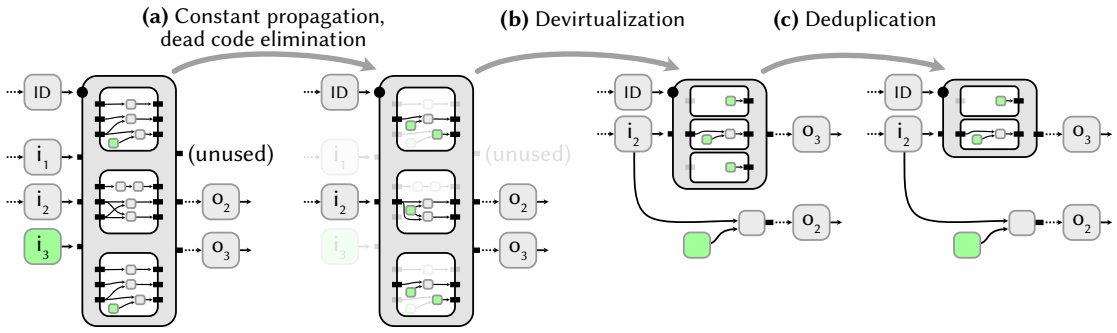


Fig. 4. When Dr.Jit encounters a polymorphic method call, it traces the implementation of all reachable instances and performs multiple optimizations. In the example shown above, a call with inputs (i_1, i_2, i_3) returns outputs (o_1, o_2, o_3). The variable i_3 is a constant literal, and the surrounding program only references outputs o_2 and o_3. **(a)**. Dr.Jit propagates the constant into the captured sub-traces, while at the same time eliminating dead code across the call boundary. **(b)**. In this example, all sub-traces perform the same computation to produce o_2, and the computation is subsequently *devirtualized*, i.e., moved out of the call. **(c)**. Finally, two of the sub-traces are found to be identical and only produce a single function definition during kernel assembly.

```
while loop(si.is_valid() & (i < 128)):
    # Sample from cosine-weighted hemisphere
    sin_phi, cos_phi = dr.sincos(rng.next_float() * dr.two_pi)
    sin_theta_2 = rng.next_float()
    wo_local = Vector3f(cos_phi * dr.sqrt(sin_theta_2),
                        sin_phi * dr.sqrt(sin_theta_2),
                        dr.sqrt(1 - sin_theta_2))
    # Rotate sample into shading frame and spawn ray with length 1
    ray_2 = si.spawn_ray(si.sh_frame.to_world(wo_local))
    ray_2.maxt = 1
    # Count unoccluded rays and increase the iteration count
    result[~scene.ray_test(ray_2)] += 1.0
    i += 1
```

Finally, we reshape and store the result as an image (right). It is at this point that the actual computation takes place within a megakernel containing all steps except for x that was previously evaluated.

```
from mitsuba import Bitmap
# Arbitrary-rank float32 tensor
from drjit.cuda import TensorXf
# Reshape/scale
tensor = TensorXf(result / 128,
                  shape=(1024, 1024))
Bitmap(tensor).write('out.exr')
```

3.2 Discussion

Following this example, we are in a better position to review high-level ideas of Dr.Jit and contrast it with other approaches.

1. Dr.Jit is principally a framework for tracing large amounts of embarrassingly parallel computation.

Tracing constitutes Dr.Jit's only mode of operation and takes place all the time. Tracing must be highly efficient, since an individual differentiation step can create millions of temporary JIT variables.

2. Dr.Jit captures the rendering process without intermediate evaluation, returning the image in the form of a trace describing the computation needed to produce it.

This ultimately enables megakernel compilation and is somewhat specific to stateless Monte Carlo methods. We did not study methods that build data structures like photon maps or irradiance caches.

3. Tracing preserves control flow like loops and polymorphism.

Backend compilation time in OptiX and LLVM tends to grow super-linearly with kernel size. Preserving control flow avoids the creation of an immensely large unrolled program that breaks the backend.

4. Dynamic compilation enables aggressive specialization and simplification of the program representation.

We will shortly see how a complete trace enables scene-specific specialization and interprocedural optimization.

Array programming. Dr.Jit shares similarities with array programming frameworks like *PyTorch*, *TensorFlow*, and *JAX* that are commonly used for differentiable programming. Like Dr.Jit, some of them can trace loops and indirect calls and compile them into fused kernels. We implemented a small renderer using JAX [Bradbury et al. 2018] and XLA [Google 2017] to run comparisons but were unable to obtain useful data, as compilation timed out on nontrivial examples.

Details of how loops and polymorphic operations are traced can greatly impact compilation and runtime performance, and we further investigated those two steps in microbenchmarks shown in Appendix A. Our take-away message from these experiments was that the large amount of arithmetic in PBDR workloads is unusual in the array programming setting designed around neural computations built from comparatively few arithmetically intensive steps. Rich ML-centric IR and tensorial optimizations can be highly effective in ML workloads, but they are costly when used to render images.

3.3 Optimizations

Dr.Jit performs optimizations to improve code generation. This may appear counter-intuitive: LLVM and OptiX are themselves sophisticated optimizing compilers, hence standard optimization passes would be subsumed by the backend compilation step. We focus on an important exception: *dynamic dispatch* presents an opaque boundary that breaks interprocedural optimizations in LLVM/OptiX.

Recall the large intersection data structure in our running example, of which only a few fields were used. Since we are already JIT-compiling, why not generate specialized code that omits unreferenced fields along with the computation needed to produce them?

This train of thought leads to a set of global optimizations illustrated in Figure 4: first, we propagate constant literals into the call, where it may trigger simplifications (this is especially effective in

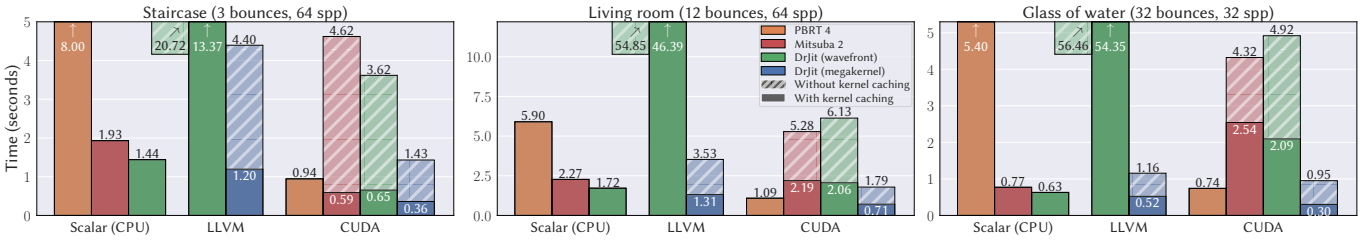


Fig. 5. We compare the performance of the combination of Mitsuba 3 and Dr.Jit with and Mitsuba 2 [Nimier-David et al. 2019] and PBRT 4 [Pharr et al. 2020], rendering three scenes of varied complexity that are shown in Figure 7. Results for both megakernel and wavefront-style evaluation are provided for Dr.Jit, where hatched bars indicate the one-time backend compilation cost (the actual kernel runtime is listed just below). PBRT is statically compiled and does not have this overhead. The performance figures demonstrate that OptiX and LLVM megakernels produced by Dr.Jit achieve state-of-the-art performance.

differentiated programs that tend to propagate many zero-valued derivatives). Removing unreferenced outputs removes unreferenced inputs, which may further cascade into the surrounding program. We also devirtualize computation that is identical in all sub-traces.

These optimizations are easily performed while tracing: constant propagation happens automatically, dead code elimination follows from variable reference counting, and deduplication builds on value numbering: if all sub-traces output a variable with the same ID, it can be moved out of the call. Finally, identical traces can be collapsed into a single subroutine. This optimization is important for rendering, where scenes often contain thousands of objects of the same type.

However, type equivalence does not imply that two instances will always perform the same sequence of operations. Consider an *ubershader* material like Disney's Principled BSDF [Burley 2012, 2015] that could expand into opaque or thin versions with different subsets of the underlying ∼11 parameters active in each instance. Textured parameters could be procedural, driven by mesh attributes, 2D bitmaps or 3D volumes with various wrapping or filtering modes. In a spectral renderer, this could furthermore involve custom spectral profiles or upsampling steps. A subset of these parameters may require derivative tracking, while others are static. We make no assumptions and trace every reachable instance, which generally produces many identical functions that can then be safely merged.

3.4 Additional features

Vectorization. The LLVM backend of Dr.Jit generates vectorized IR targeting the SIMD instruction set of the host. This means that operations like loops, function calls, and ray tracing process 4-16 elements at a time. Function calls may require several iterations to handle all elements in the presence of divergence. Dr.Jit automatically performs such transformations, while adding masking similar to the *ISPC* compiler [Pharr and Mark 2012]. Function deduplication discussed in Section 3.3 is important for vectorized execution on both CPUs and GPUs because it can significantly reduce divergence.

Closure generation. Instance-specific attributes like the roughness parameter of a BRDF model require special handling during tracing. Dr.Jit automatically detects such implicit dependences and adds them to a *function closure*. Appendix B provides details on this.

Loop optimizations. Besides polymorphism-related optimizations, Dr.Jit also optimizes loop constructs by removing invariant or unreferenced state variables. LLVM and OptiX can in principle perform similar optimizations, but polymorphism within loops can sometimes impede them. We leverage Dr.Jit's global view across loops and indirections to remove redundancies.

Wavefront-style evaluation. Several flags control the tracing behavior of Dr.Jit: for example, it can either trace loops or execute them using a sequence of wavefront-style kernels. Similarly, it can trace polymorphic function calls or group the wavefront by instance and launch a maximally coherent kernel per group. Besides full megakernel- or full wavefront-style evaluation resembling Enoki [Jakob 2019], this provides an intermediate configuration with wavefront loops and traced calls that can be surprisingly effective.

Side effects. Several of Dr.Jit's builtin operations like scatter operations and atomic scatter-reductions mutate existing arrays. Read access or arithmetic involving variables marked as being affected by queued side effects trigger a kernel launch to materialize them.

3.5 Results

We now turn to a first set of results that examine *primal* rendering performance, comparing Dr.Jit to two open source renderers with wavefront-style evaluation: Mitsuba 2 [Nimier-David et al. 2019] using the Enoki library [Jakob 2019], and PBRT 4 [Pharr et al. 2020].

The three benchmark scenes are shown in Figure 7: *Staircase* contains 749 shapes and 24 BSDFs and simulates low-order scattering, while *Living room* uses longer paths with 12 interactions. *Glass of water* simulates up to 32 dielectric refractions and reflections. Good performance in these benchmarks requires efficient handling of loops for interreflection and dynamic dispatch to many scene objects.

Experiments throughout this article were performed on an AMD Ryzen Threadripper 3990X server (64+64 virtualized cores, 128 GiB RAM) with an NVIDIA RTX A6000 GPU (45.6 GiB RAM). The vendor-specified thermal design power of CPU (280W) and GPU (300W) are roughly matched to enable a fair comparison between these two very different processor architectures. On the software side, we use Linux kernel 5.13, LLVM 10, and NVIDIA driver 510 with a vendor-provided fix for a performance regression found during development (this fix will be part of future driver releases). We follow standard benchmarking practices like disabling mitigations for side-channel attacks, reporting the median of multiple (5) runs, and pre-allocating 2 MiB *huge pages* used by Embree [Wald et al. 2014] and Dr.Jit.

Three groups of columns in Figure 5 contrast statically compiled one-ray-at-a-time rendering (*scalar* C++ code) with execution via LLVM, and OptiX. Dr.Jit results are presented in both wavefront and megakernel mode. Our system compiles a specialized megakernel

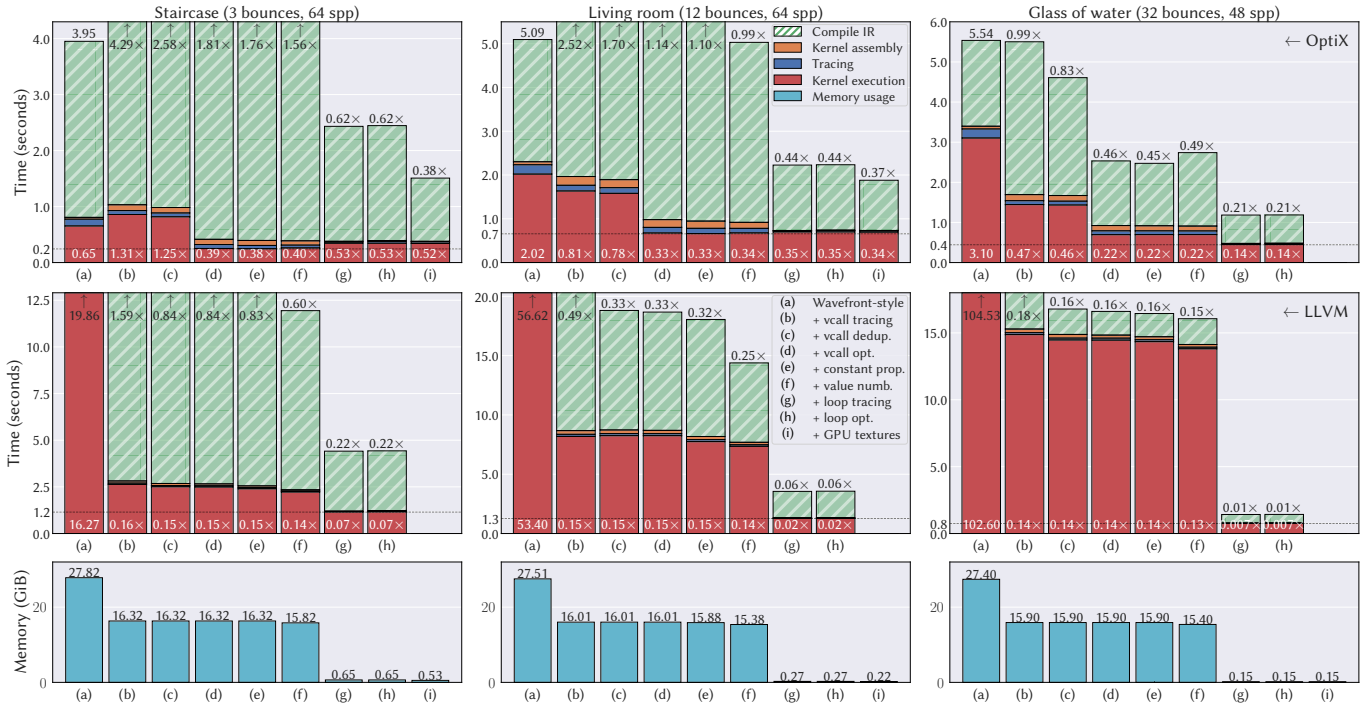


Fig. 6. We investigate the effect of different optimizations in Dr.Jit using the same set of scenes (columns). The rows show OptiX and LLVM runtime costs followed by peak memory usage that is identical in both backends. Stacked bars indicate the time spent on backend IR compilation (hatched), kernel assembly (orange) and tracing (blue) within Dr.Jit, as well as kernel execution (red). The leftmost bar **(a)** is a wavefront baseline that "unrolls" loops and polymorphism into separate kernels that communicate through device memory. Going from left to right, we then successively enable **(b)** compilation of polymorphism into subroutines, **(c)** deduplication of subroutines containing identical code, **(d)** global polymorphism-aware optimizations (Section 3.3), **(e)** constant propagation, **(f)** local value numbering, **(g)** tracing loops instead of unrolling them, **(h)** loop state optimizations, and **(i)** hardware accelerated texture lookups (GPU only).

(or series of smaller kernels) for each scene, which incurs a one-time overhead represented by hatched bars; the solid portion below indicates performance once the system is warmed up. We applied minor optimization to Mitsuba 3 while incorporating Dr.Jit, such as marking primary rays as coherent in Embree. These also benefit the renderer's statically compiled scalar mode, which we represent using the green Dr.Jit bar in the "Scalar (CPU)" column.

Dr.Jit performance compares favorably, achieving geometric mean speedups of 3.70 × (vs. Mitsuba 2) and 2.14 × (vs. PBRT 4) on the GPU. On the CPU, vectorized LLVM megakernels produced by our system outperform scalar rendering in all cases.

Compilation into wavefronts produces programs that must repeatedly read and write vast amounts of data, imposing a burden on the processor's memory subsystem. Difference in memory bandwidth on the CPU (~95 GiB/s) and GPU (~715 GiB/s) as well as latency hiding-capabilities in the latter lead to marked differences between architectures in this case. Wavefront-style execution reduces GPU performance by a factor of 2.0 – 7.7×, while the same experiment on the CPU can increase cost by a factor of more than 100×.

Figure 6 investigates the effect of individual optimizations starting from a wavefront-style baseline resembling the operation of the Enoki library (column **a**). Memory usage is significant in this configuration (bottom row). While this could be reduced by launching many smaller wavefronts, it would not address the fundamental issue that a large amount of data will need to be read and written.

A high level observation is that tracing and kernel assembly (orange and blue bar regions) only constitute a small portion of the total computation time. The precise amount tends to be lower when compiling megakernels and higher for wavefronts requiring multiple rounds of tracing and assembly. Backend compilation time to transform IR into machine code can be significant (longer than the kernel execution itself), which can make the system unsuitable for certain use cases and emphasizes the importance of caching mechanisms.

Mapping polymorphic function calls to subroutines in **(b)** drastically reduces runtime on the CPU while increasing compilation time on both backends; subroutine deduplication in **(c)** reverts most of the growth in compilation time and also gives a speed boost thanks to reduced divergence under vectorized execution.

Removing unreferenced computation, arguments, and return values from polymorphic method calls in **(d)** has a surprisingly large effect in the OptiX backend, where it improves performance by an average factor of 2.5×, while having essentially no impact on the CPU. Modern superscalar CPUs can absorb a certain amount of redundant computation by issuing multiple instructions per clock cycle. We speculate that these architectural features along with the lower cost of data exchange through the stack could be responsible

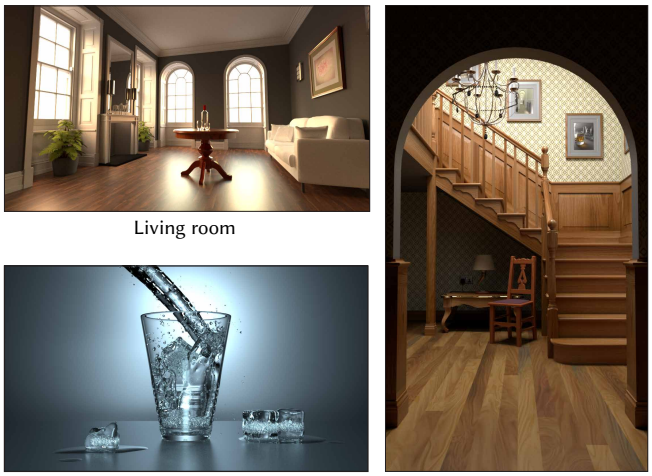


Fig. 7. Scenes from Bitterli's [2016] rendering resources used in benchmarks.

for the modest benefit of this optimization (though it will prove to be more effective when applied to differential kernels in Section 4).

Loop tracing in **(g)** reduces memory traffic to a minimum, which greatly benefits CPU execution, while having an inconclusive effect on the GPU. We speculate that loops may exacerbate the conversion into an OptiX state machine, whose nodes must exchange continuation state [Parker et al. 2010]. Loop optimizations in step **(h)** remove ~65% of state variables (details in Section 4.10), but this appears to be mostly subsumed by the backend's optimization passes.

Performing GPU texture lookups using hardware texture mapping units in **(i)** yields small but measurable runtime improvements on the order of 1% when compared to software-based interpolation. Compilation time improves by up to 30% when the scene contains many textured materials (e.g., *Staircase*), since IR operations related to wrapping and interpolation can be removed.

3.6 Limitations

The following are notable limitations of the tracing approach.

The need to evaluate. Suppose we wanted to run the ambient occlusion integrator multiple times to produce uncorrelated renderings. A naïve attempt to do so by reusing the pseudorandom number `rng` could lead to a problem where caching breaks down, requiring repeated backend compilation of kernels that become bigger and slower over time. This happens because we never asked the system to materialize the RNG state into data; all RNG updates from prior rendering steps are still part of the trace and spill into the current kernel. Calling `dr.eval(result, rng)` solves this problem, though the need for this step can be non-intuitive to new users.

Recursion. Dr.Jit can trace dynamic dispatch to functions that themselves perform dynamic dispatch, which is, e.g., needed to handle object transformations involving instanced geometry. Cycles including self-recursive calls are not permitted, as the system would then trace the same code repeatedly without knowing when to stop.

Debugging. We provide two ways to investigate the behavior of a program: the user can set Dr.Jit to wavefront-style evaluation (Section 3.4), set breakpoints, step through the program, and investigate variable contents. Megakernel compilation is less flexible: the only available option in this case is to print from the device via `dr.printf_async()`, which will produce output when the kernel runs later. More work will be needed to develop better debugging primitives for this unusual way of performing computation.

4 DIFFERENTIATION

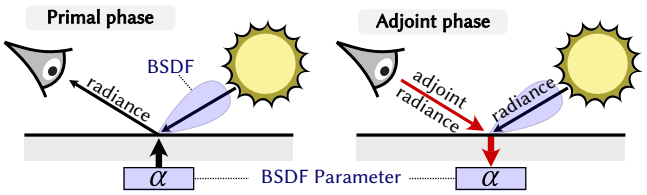
Following this presentation of Dr.Jit's computational foundation, we now turn to differentiation starting with a review of PBDR techniques and ways in which they constrain the system.

4.1 Physically Based Differentiable Rendering

A typical rendering algorithm like path tracing [Kajiya 1986] propagates *radiance* from light sources to the sensor. To do so, it loops over consecutive path vertices starting from the sensor. Many iterations might be necessary when the scene contains scattering media.

Such loops are unfortunately a nuisance during differentiation, as each iteration generates intermediate state that must be reconstituted to compute reverse-mode derivatives (Section 2.2). In the case of a path tracer, millions of Monte Carlo samples will run the loop in parallel for an unpredictable number of iterations, hence large quantities of memory would need to be provisioned conservatively to store the resulting intermediate state. In general, we find that the automatic derivative of a loop is almost never satisfactory; the user should instead contribute domain-specific knowledge to implement an equivalent and more efficient *custom adjoint*.

Radiative Backpropagation (RB) [Nimier-David et al. 2020] is such a custom adjoint. Using physical reciprocity, it replaces the derivative of the simulation with an equivalent simulation of *differential* radiation ("*adjoint radiance*"), removing the need to store loop state. Differentiating the image loss initially yields adjoint radiance in image space, where it describes how pixels in the rendered image should change to reduce the loss. Like a video projector, the camera then emits this signed radiation into the scene, where it scatters just like normal light. In contrast to primal rendering that queries the scene representation using *reads*, RB is a *write*-heavy method: whenever adjoint radiance encounters a surface with differentiable parameters, the method accumulates a contribution into the local parameter gradient, for example to optimize α below.



The derivative with respect to α is also proportional to the amount of incident radiance (a surface in darkness generates no gradients). This illustrates the main issue with RB: it requires an intertwined simulation of both adjoint radiance *and* primal radiance, which leads to a costly algorithm with quadratic runtime complexity.

Path Replay Backpropagation (PRB) [Vicini et al. 2021] fixes this problem using a two-pass approach. Its first pass generates a set of Monte Carlo samples and stores data consumed by a subsequent pass.

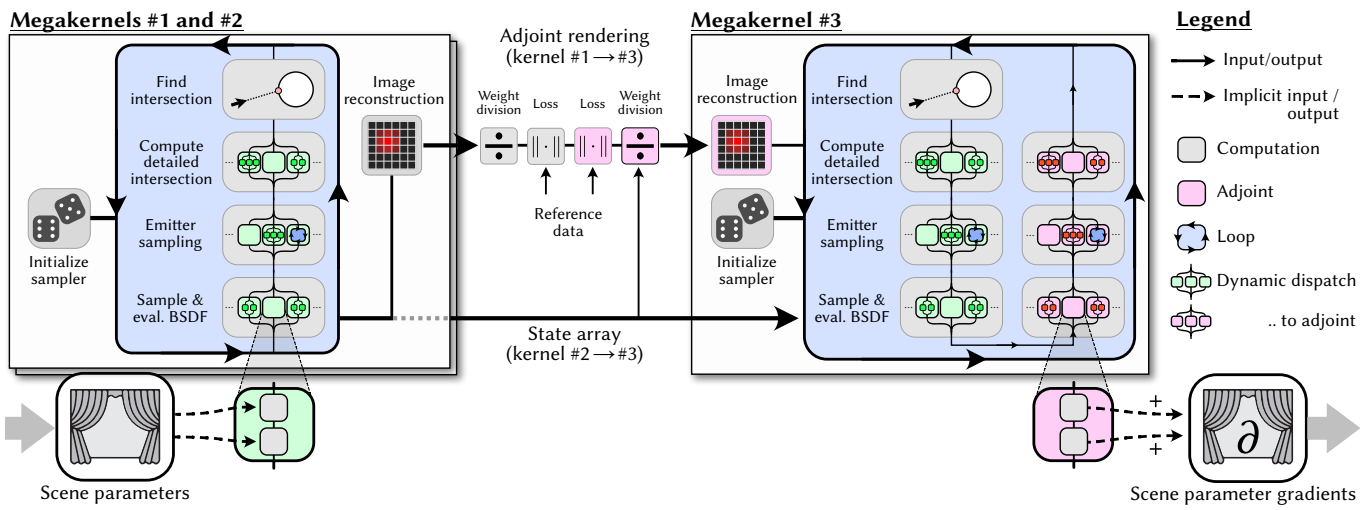


Fig. 8. The anatomy of a recent physically based differentiable rendering method. This diagram illustrates a partition of the major components of *Path Replay Backpropagation* (PRB) [Vicini et al. 2021] into a set of three self-contained megakernels that each solve a Monte Carlo integration problem. Megakernel #1 is a standard path tracer that dynamically dispatches function calls to scene objects (shapes, emitters, materials, etc.) with an implicit dependence on scene parameters. The path tracer uses an image reconstruction filter to scatter weighted samples values into an output buffer with subsequent weight division [Pharr et al. 2020], which produces a *primal* rendering that is passed into a loss function to quantify the quality of the current iterate. Meanwhile, a second path tracing megakernel performs the same computation once more with a different pseudorandom seed, providing a per-sample state array to a final *adjoint* megakernel #3. This kernel contains adjoint versions of all steps and is responsible for accumulating scene parameter gradients.

This second pass regenerates the same set of samples, exploiting the precomputed data and arithmetic invertibility to recover the incident radiance at every vertex. With this information, the desired gradients can be accumulated. A complete PRB optimization step actually computes one further *decorrelated* [Gkioulekas et al. 2016] primal rendering that is provided to the image loss function, so there are *three* integration steps in total. Figure 8 illustrates the high-level architecture and a suggested decomposition into megakernels.

There are other important problems to consider besides efficiency. Rendering algorithms evaluate integrals containing visibility-related discontinuities. When differentiated parameters affect visibility (and thereby the position of the discontinuities), the resulting derivatives tend to be severely biased, which breaks shape reconstruction unless special precautions are taken. Figure 9 visualizes bias in a simple forward mode example. Techniques to remove bias include *edge sampling* [Li et al. 2018a] *reparameterizations* [Loubet et al. 2019; Bangaru et al. 2020], and *path space sampling* [Zhang et al. 2020].

Finally, differentiation changes the integrals computed by the simulation. Sampling strategies designed for a primal integrand may perform poorly when applied to its derivative, in which case derivative-aware strategies may be necessary [Zeltner et al. 2021].

Our goal in the remainder of this paper is to establish a solid foundation for these and future PBDR methods. We must, however, limit the scope somewhat and therefore mainly focus on the example of a reparameterized path replay backpropagation integrator.

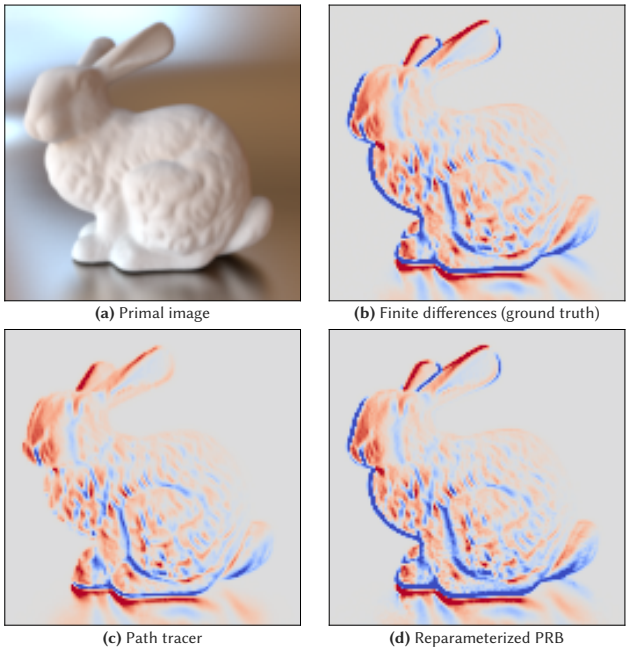


Fig. 9. Parameters that affect the visibility in a scene produce biased derivatives unless extra precautions are taken. This figure shows the forward derivative of a Stanford bunny on a metallic floor with respect to translation. **(a)** Primal rendering. **(b)** Reference. **(c)** Naïve derivative of a path tracer. Bias manifests in the form of incorrect silhouette gradients and reflections in the floor. **(d)** Reparameterizing integrals removes the discrepancy.

4.2 Objectives

A central tenet of Sections 1 and 3 was that the system should never partition Monte Carlo integration across multiple kernels to avoid costly inter-kernel communication. Tracing also needed to

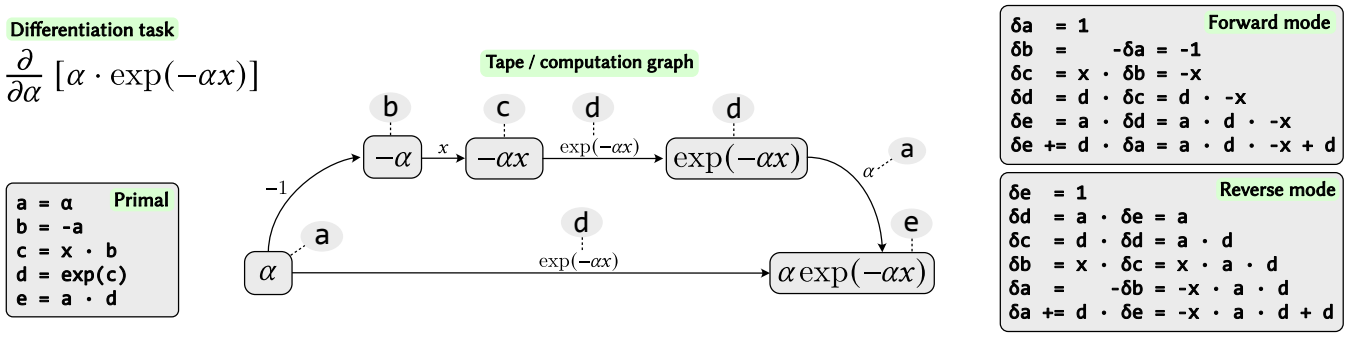


Fig. 10. An example use of tape-based forward/reverse-mode AD to differentiate the exponential density $\alpha \exp(-\alpha x)$ with respect to α. The variable x does not carry derivatives (it is *detached*) and could, e.g., represent a Monte Carlo sample position. Primal evaluation of this expression (bottom left) generates a sequence of temporaries (a b, c, d, and e) and replicates the dependency structure of their computation in a tape/computation graph (middle). Each edge in this graph carries a weight indicating the sensitivity of the target node with respect to perturbations of the source node. Often, these edge weights are simply variables of the primal program, which is indicated by ellipses. AD associates a derivative variable with each temporary and program variable (e.g., a and δa). Forward- and reverse modes (right) traverse the tape as indicated by their name, propagating derivatives scaled according to the edge weights. The result is equivalent, though the efficiency of the two modes can drastically differ when the computation has many inputs or outputs.

preserve loops and polymorphic calls to avoid creating immensely large unrolled kernels that would be challenging to compile.

The same objectives also apply to the generation of *differential megakernels*. Dr.Jit must be able to trace an algorithm like PRB and compile its Monte Carlo integration steps into self-contained kernels with preserved control flow. This introduces several new challenges related to differentiation:

- PRB performs a differentiable computation at every scattering interaction and then invokes AD to back- or forward-propagate associated derivatives. Dr.Jit must convert these high-level AD operations into code that can be included in a megakernel.
- The differentiable computation contains many polymorphic function calls. Their derivative should also be a polymorphic call targeting generated forward or reverse-mode derivative versions of the primal function implementations.
- Derivatives should flow through typical preprocessing steps like the computation of smooth normals or MIP maps. However, data dependencies prevent the evaluation of these derivatives within the same megakernel. Dr.Jit must partition the AD task into multiple phases with efficient information exchange.
- Dr.Jit must support derivative-related transformations that transcend raw differentiation. For example, it should be possible to introduce complex transformations like reparameterizations.

Dr.Jit once more addresses these requirements using an approach based on tracing. This tracing of differentiable computation occurs at a higher architectural level and depends on the JIT compiler as foundation, which means that the combined system traces at *two levels simultaneously*. This combination is harmonious: besides fulfilling the stated requirements, it enables Dr.Jit to dynamically specialize differential algorithms to the scene and problem statement, while removing redundant steps introduced by the AD transformation.

A caveat worth noting here is that Dr.Jit does not free the developer from thinking about subtle details of the differentiation process. It facilitates—but does not automate—the introduction of the previously mentioned physical and mathematical "reversibilities".

The remainder of this section reviews tracing AD and explains how it can be adapted to address the stated challenges. Unless noted otherwise, subsequent use of the word *trace* will now refer to a *(Wengert) tape*, or *computation graph* that captures operations for subsequent derivative propagation.

4.3 Tracing with dynamic compilation

Section 2 introduced two high-level flavors of AD: *tracing* and *source transformation*. Dr.Jit lies somewhere in between: it ingests computation using tracing, but the combination with dynamic compilation and function-level differentiation produces output resembling that of source transformation AD.

Figure 10 shows a simple example of tracing-based AD. The tape associates every primal variable x with a differential version δx and records inter-variable dependencies. Subsequent tape traversal computes the differentiable variables, which produces a flurry of operations referencing primal program state (Figure 10 right). A subtle but important detail is that this derivative-related computation is itself *traced* by the underlying JIT compiler. Said overly dramatically, differentiation destroys the AD data structures, which leaves a residue of ordinary computation. This residue is traced by the JIT-compiler one level below so that it can run as part of a (mega-) kernel at some later point. The actual destruction of the tape turns out to be immaterial in this process; it is often useful to retain it when multiple AD traversals are needed. This approach is not a contribution of Dr.Jit and also underlies other hybrid AD systems, e.g., CppADCodeGen [Leal and Bell 2017].

It is interesting to note that this combined system supports *checkpointing* without having put any intentional effort into realizing such a feature. Consider an unrolled loop with intermediate evaluation:

```
for i in range(1000):
    data = f(data) # 'f' represents a potentially complex
    dr.eval(data)  # transformation of 'data'.
dr.backward(data)  # Now, backpropagate through all steps
```

The AD layer will reference intermediate steps and temporaries produced by the function `f` to enable subsequent differentiation, and these must be recomputed. The JIT layer knows how to compute these needed variables, and it will do so from the last evaluation point that therefore takes the role of a checkpoint. Checkpointing is not ideal for PBDR workloads due to the size of even a small number checkpoints, though it does help when differentiating rendering techniques that have not been adapted for efficient differentiation.

4.4 Custom adjoints

Like other AD systems, Dr.Jit supports functions with user-provided forward-, and reverse-mode derivatives ("custom adjoints"), by extending an interface named `dr.CustomOp`. When an AD traversal encounters a `CustomOp` on the tape, it will invoke its `forward()` or `backward()` callbacks to convert function input derivatives into output derivatives, or vice versa. In contrast to other AD systems that consider *explicit* function arguments and return values, Dr.Jit also tracks *implicit* inputs or outputs. This refers to dependencies on global state that only become apparent while running the function. Tracking implicit reads and writes ensures that the global flow of derivatives is correctly represented and will be useful shortly.

4.5 Differentiating polymorphism

With this infrastructure in place, we are ready to discuss polymorphism. We will consider the following method call, where `obj` refers to a Dr.Jit array of instances that implement the function `func`.

```
out = obj.func(arg_1, arg_2, ..)
```

The derivative of such a polymorphic method call, is *another* polymorphic method call to JVP and VJP (Section 2.2) versions of the original set of functions. Dr.Jit generates these dynamically as needed. To detect this need, it wraps use of polymorphism like the example above into a dynamically generated `dr.CustomOp`. Suppose that the `CustomOp.backward()` callback is now invoked during an AD traversal, which indicates that the system needs to propagate `dr.grad(out)` to `dr.grad(arg_1)`, `dr.grad(arg_2)`, etc.

To accomplish this, the AD layer dynamically defines a VJP per method, which is simply a placeholder that calls `func` a second time, with recursive usage of AD to propagate and extract derivatives.

```
def func_vjp(grad_out, arg_1, arg_2, ..):
    dr.enable_grad(arg_1, arg_2, ..)
    result = func(arg_1, arg_2, ..)
    dr.set_grad(result, grad_out)
    dr.backward(result)
    return dr.grad(arg_1), dr.grad(arg_2), ..
```

The AD layer then issues a polymorphic method call to these newly defined functions. To realize this method call, the JIT layer traces each possible target, which runs all VJP placeholder functions, materializing the differentiation into ordinary traced computation.

Optimizing polymorphic derivatives. Suppose that one of the return values of `obj.func()` does not carry derivatives. Dr.Jit will devirtualize this zero-valued derivative following Section 3.3, enabling optimizations on the caller's side. Zero-valued input derivatives are propagated into the call, simplifying call targets using constant propagation. The effect of these simple steps can be significant.

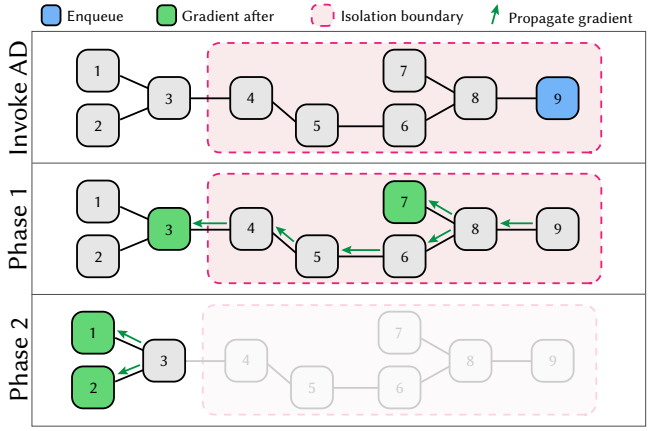


Fig. 11. Dr.Jit provides three different AD *scopes*. The first is an *isolation boundary*. Wrapping computation into such a scope restrains the process of derivative propagation: only edges within and just across the isolation boundary may be traversed, while others are postponed until the scope is destructed. Dr.Jit uses this feature to ensure correct generation of a differential megakernel that is embedded within a larger differential calculation.

Automatic discovery of implicit dependences. Section 3.4 and Appendix B observed that the functions being differentiated are generally *closures* that implicitly depend on further variables from a surrounding environment, for example BRDF or texture parameters. This creates differentiable dependences that must be tracked by the AD system. Our system monitors variable accesses during the primal `CustomOp` evaluation and automatically registers implicit variable dependences with the previously discussed mechanism.

The reverse mode derivative of a method call with implicit reads will issue atomic scatter-reductions to update scene parameters gradients. This is not just an odd corner case; it constitutes the main way in which scene parameter gradients are generated.

4.6 Isolation boundaries

PRB performs a differentiable computation at every scattering interaction, followed by a backpropagation step. In the example below, `Li`, `Lr` and `δL` refer to incident, reflected, and adjoint radiance.

```
while loop(depth < max_depth):
    # ... Compute surface interaction 'si '...
    Lr = Li * si.bsdf().eval(si, wo) # Reflected radiance from 'wo'
    dr.backward(δL * Lr) # Backpropagate product
```

This differentiation step is likely part of a larger computation including preprocessing steps that must also be differentiated. Suppose the that BSDF queries a texture, whose MIP levels were computed using successive reductions: the derivative of this process must upsample gradients back to the finest level, requiring multiple passes with intermediate data exchange, which cannot be done within the megakernel being compiled. Dr.Jit provides *isolation boundaries* (Figure 11) to *postpone* such problematic steps.

```
with dr.isolate_grad():
    # .. temporarily isolate outside world from AD traversals ..
```

Loops and polymorphic calls implicitly create an isolation boundary.

4.7 Reparameterizations

Parameters influencing the position and shape of scene geometry produce bias when differentiated (Figure 9) unless the Monte Carlo integrals within the rendering algorithm are reparameterized. Reparameterizations counteract parameter-dependent silhouette motion so that discontinuities are *frozen* when observed within spherical integrals. We expose reparameterizations through a `CustomOp`-based abstraction to isolate their specifics from the PBDR integrator. A reparameterized ray intersection then reduces to two lines:

```
ray.d, det = reparameterize(ray)
si = scene.ray_intersect(ray)
```

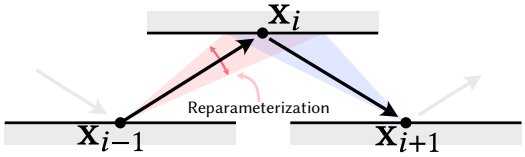
where `det` is Jacobian determinant of the change of variables.

Reparameterizations are a somewhat strange concept. During primal rendering, they reduce to the identity: `reparameterize()` simply returns the input ray direction along with a Jacobian determinant of 1. When differentiated, they expand into intricate operations (Figure 12) that entail tracing auxiliary rays, weighting them, and using AD to scatter gradients into intersected shapes. Usage of AD is therefore recursive: a top-level `render()` function launches PRB (a `CustomOp`) when differentiated, which differentiates contributions in a traced loop and thereby evaluates the derivatives of various system operations including parameterizations (a `CustomOp`), which launches differentiable ray tracing operations (a `CustomOp`). Another level of recursion may take place in the presence of instancing.

```
render(scene)
└→ Path Replay Backpropagation (CustomOp)
   └→ Reparameterization (CustomOp)
      └→ Shape.compute_surface_interaction (CustomOp)
```

4.8 Selective evaluation of partial derivatives

PRB consists of a loop with nested use of AD to compute light path derivatives one vertex at a time. Reparameterizations introduce an extra complication by perturbing the positions of these path vertices.



Moving $\mathbf{x}_i$ may change the BSDF at $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$. A reparameterized integrator must account for this dependence when visiting the reparameterized vertex $\mathbf{x}_i$ to ensure that all partial derivative terms are present. However, separate differentiation of these two BSDF terms would also generate scene parameter derivatives, e.g., with respect to the albedo of $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$. This is undesirable when those derivatives were already generated elsewhere: we require a way of only differentiating the dependence on the position of $\mathbf{x}_i$.

To address these and similar issues, Dr.Jit's AD layer keeps track of a set of variables Ω for which derivative tracking is currently enabled. Two additional *AD scopes* modify this set, by setting it to the empty set $\emptyset$, its complement $\emptyset^c$, or by adding and subtracting variables from the current set. A PBDR algorithm using these abstractions repeatedly enters and leaves scopes to control what derivative terms should be generated.

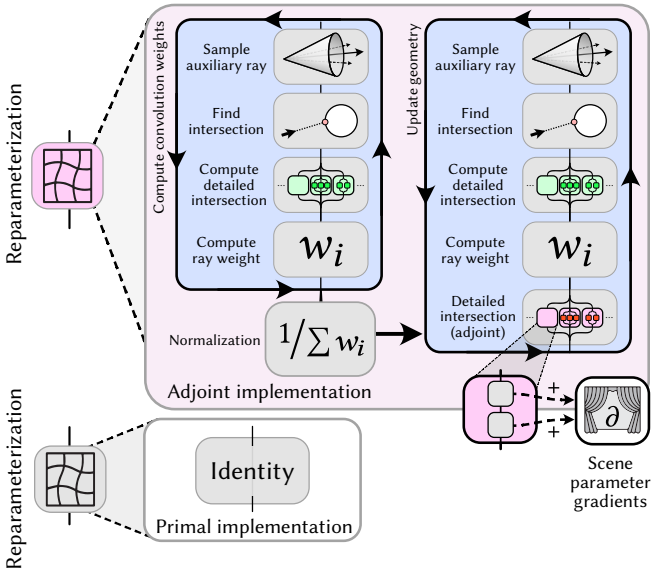


Fig. 12. To address the visibility-induced bias shown in Figure 9, rays must be *reparameterized* [Loubet et al. 2019]. In primal computation, reparameterizations have no effect and reduce to the identity. When differentiated, they trace auxiliary rays to construct a warp field [Bangaru et al. 2020] that counteracts silhouette motion. The reverse-mode derivative of this warp field scatters scene parameter gradients into intersected geometry.

```
with dr.suspend_grad():                  # Ω = ∅
    with dr.resume_grad():               # Ω = ∅ᶜ
        ray.d, det = reparameterize(ray)
        si = scene.ray_intersect(ray)
    # .. detached sampling steps ..
    with dr.resume_grad(ray.d):          # Ω = {ray.d}
        # Only account for directional derivatives at 'si_prev'
        L += Li_prev * si_prev.bsdf().eval(si_prev, ray.d)
    # .. other steps ..
    with dr.resume_grad():               # Ω = ∅ᶜ
        dr.backward(δL * L) # Backpropagate through all terms
```

4.9 AD tape surgery

Implementations of PRB involve mysterious-looking steps like

```
Lr *= bsdf_value_diff / bsdf_value
```

where the `bsdf_value` and `bsdf_value_diff` variables have the *same value*. What purpose do they serve?

In this example, `Lr` stores the reflected radiance obtained using a *detached* BSDF sampling strategy [Zeltner et al. 2021], which means that the process of importance sampling was not differentiated. To generate nonzero derivatives, the snippet above changes a factor within the expression in `Lr` (specifically, `bsdf_value`) so that it is *attached*, i.e., recomputed *with* AD-based derivative tracking.

A problem with this approach is that it introduces unnecessary extra computation in both primal and adjoint programs. We provide the `dr.replace_grad(a, b)` function that can be used to accomplish this goal more efficiently. It combines the primal value of 'a' with the AD trace of 'b', which changes the previous example to

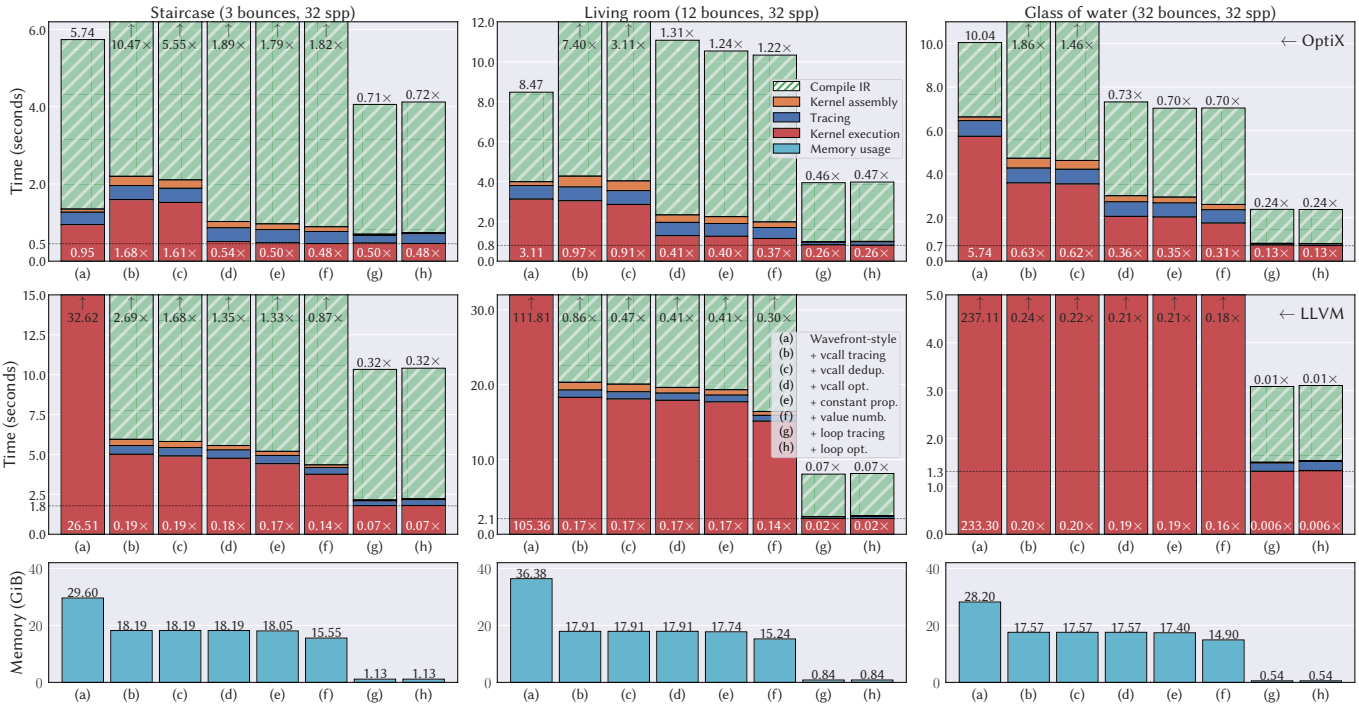


Fig. 13. Reverse-mode differentiation benchmark of *non-reparameterized* Path Replay Backpropagation analogous to Figure 6. We differentiate the rendered output of three scenes with respect to albedos and albedo textures, analyzing performance and the effect of different optimizations in Dr.Jit. The rows show OptiX and LLVM runtime and peak memory usage (identical). Stacked bars indicate the time spent on backend IR compilation (hatched), kernel assembly (orange) and tracing (blue) within Dr.Jit, as well as kernel execution (red). The wavefront baseline **(a)** on the left "unrolls" all use of loops and polymorphism into separate kernels that communicate through global memory. Going to the right, we successively enable **(b)** constant propagation, **(c)** compilation of polymorphism into subroutines, **(d)** subroutine deduplication, **(e)** optimization of polymorphic calls, **(f)** local value numbering, **(g)** loop tracing, and **(h)** loop optimizations.

```
Lr *= dr.replace_grad(1, bsdf_value_diff / bsdf_value)
```

We also use this feature to turn texture lookups performed using GPU texture mapping units (TMUs) into differentiable operations by attaching them to the AD graph of a software-based lookup.

4.10 Results

We now present results showcasing the combination of differentiation and dynamic compilation. We not pursue complex application scenarios; our focus is purely on the structure of the computation and the performance of the system on such differential workloads.

Reverse-mode differentiation benchmark. Using the same set of scenes as before (*staircase*, *living room*, *glass of water*), we now use PRB to differentiate albedo values (scalar, textured) and emitters like the environment map in the *living room* scene. The information in Figure 13 only reflects the differential portion of a gradient step (i.e., phases #2 and #3 of the partition shown in Figure 8).

Many of the observations mirror our previous discussion of primal rendering in section 3.5. On the CPU, the benefit of megakernel-style evaluation continues to be dramatic, with speedups reaching ∼166 × compared to the baseline (∼142 × in the primal benchmark).

One major change compared to the primal setting is that the computation must now evaluate the reverse-mode derivative of polymorphic calls. Consider the derivative of a function like BSDF evaluation that takes a large intersection record as input. Its derivative has even more inputs, and it additionally returns an output derivative for each primal input argument. Detecting and removing the resulting redundancies using global dead code elimination, constant propagation, and value numbering has a pronounced effect, with GPU speedups from this alone reaching 3.3 × in the *staircase* scene.

The differential megakernels enabled by the methods presented in this article generally achieve the lowest tracing and kernel assembly time, lowest compilation time, and lowest runtime besides using a minimal amount of GPU memory. This last point becomes relevant when optimizing large scene representations (e.g. 3D volumes) that consume most of the available device memory.

Reparameterized PRB benchmark. Figure 14 repeats this experiment once more, using *reparameterized PRB*. The introduction of reparameterizations and steps needed to evaluate the derivatives of adjacent vertices (Section 4.8) now leads to very large programs compared to primal rendering or non-reparameterized methods.

This time, we optimize the vertex positions of all scene geometry[3]. Again, this experiment produces no surprises and shows the effectiveness of the various optimizations. An interesting contrast between Figures 13 and 14 is the staircase-like sequence of bars in

[3]Derivatives will be biased in the *glass of water* scene containing dielectric objects. How to reparameterize through perfectly specular interfaces is an open research problem.

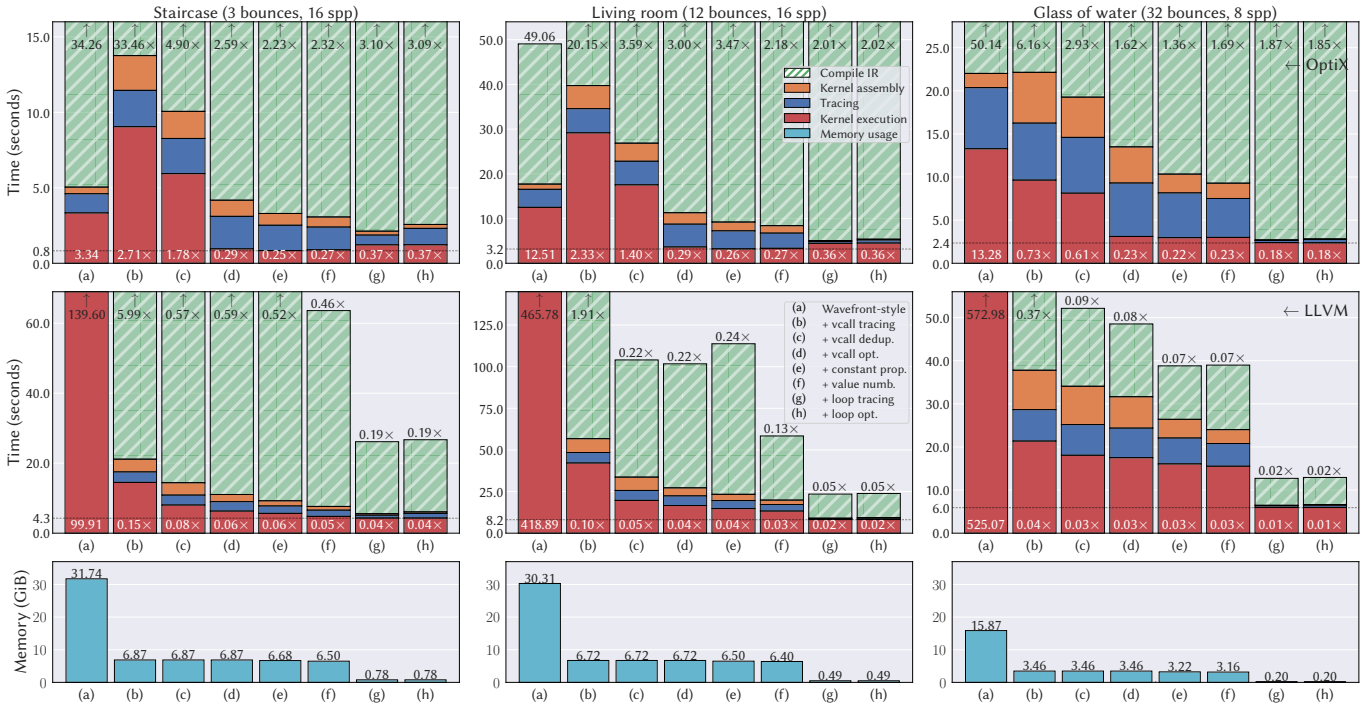


Fig. 14. Reverse-mode differentiation benchmark of *reparameterized* Path Replay Backpropagation analogous to Figures 6 and 13. Please see their captions for details on the visualization, and Section 4.10 for a discussion of this result.

the latter, which clearly shows the benefit that each of the separate steps makes in both CUDA and especially the LLVM backend (which originally had a relatively flat profile in Figure 13). Once more, polymorphism-related optimizations are very important, with dead code elimination, value numbering, and constant propagation now producing a 6.6× speedup on the OptiX backend.

The improvements of the loop state optimizations on runtime performance are relatively modest (runtime improvements of a few percent in Figure 13) compared to other steps. We believe that this optimization may be more effective in methods using *attached* Monte Carlo sampling [Zeltner et al. 2021; Vicini et al. 2021]), where large sets of derivatives must be exchanged between loop iterations.

Size reductions and size increases. Figure 15 provides another view of the effectiveness of optimizations besides compilation and runtime performance. It shows that the majority of function arguments and return values are removed regardless of the application. In reverse-mode (reparameterized) PRB, the number of removed function outputs goes up significantly, as many computed derivatives have no effect. Reparameterized PRB carries a large amount of loop state to analyze the interaction between three adjacent path vertices; the implementation of this method was carefully optimized, which explains the smaller effect in the last row.

Table 1 gives an impression of how differentiation increases the size of a rendering algorithm, as measured in the number of compiled IR operations. The data indicates a relatively stable growth of 3–4× when going from primal path tracing to (differential) path replay. This is expected: besides steps for differentiation, the method runs two separate rendering passes. When reparameterizations are added on top, a different behavior emerges: wavefront evaluation produces significant (> 30×) growth in the operation count, which our optimizations then stabilize to a factor of 7–8×.

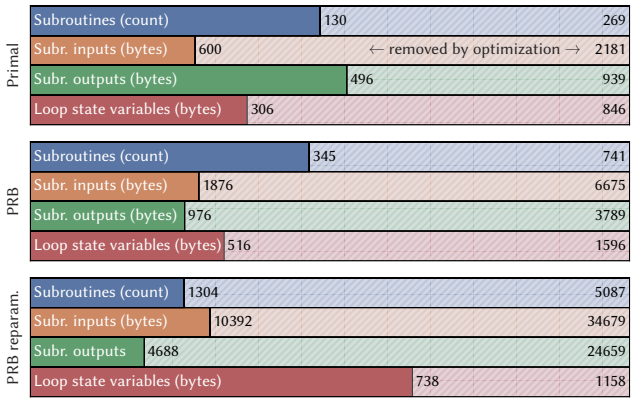


Fig. 15. Visualization of the total number of generated kernel subroutines, and the amount of data exchanged through function inputs, outputs, and loop state variables. The stated amounts are sums across the three benchmark scenes. Dashed bars indicate the proportion that is removed by the presented optimizations. The three rows showcase the differences in behavior when compiling primal rendering, PRB, and reparameterized PRB methods.

Differentiating existing methods. Dr.Jit can also differentiate standard methods that have not been designed for this. Our last experiment demonstrates this using the builtin Mitsuba (volumetric) path

tracer and contrasts its behavior to PRB. The path tracer contains a loop that Dr.Jit handles using a checkpoint per scattering event (Section 4.3). Differentiable polymorphism can still be used within each iteration, reducing memory usage and communication costs. Columns **(a)** and **(b)** show the performance of such an approach.

Differentiating with checkpoints is very memory-intensive but has the advantage that only two Monte Carlo integration phases are needed in contrast to PRB's three phases. Given a large supply of high-bandwidth memory (e.g. on the GPU), checkpointing can yield acceptable performance. The runtime cost for PRB's two-stage approach is initially higher in **(c)** (wavefront loop with polymorphism) but outpaces checkpointing once compiled to a megakernel in **(d)**. (These observations are consistent with the benchmarks reported in the original paper [Vicini et al. 2021].) Communication costs cause checkpointing to become less competitive when the differentiated computation involves long-running loops, as shown in the right example that differentiates a rendering with heterogeneous subsurface scattering and a maximum path length of 128 interactions.

5 CONCLUSION

Dr.Jit is a specialized compiler for physically based differentiable rendering algorithms, whose unique set of constraints makes their implementation using traditional means near-impossible. Dr.Jit can trace large object-oriented codebases with polymorphic indirections, while providing fine-grained control over the differentiation process that is needed to exploit physical and mathematical symmetries.

Its combination of tracing at JIT and AD levels is harmonious: by tracking the flow of primal and differential quantities globally, Dr.Jit can specialize algorithms to the problem at hand, while discarding redundant computation. Its wavefront and megakernel implementations achieve excellent performance and run on diverse hardware.

Dr.Jit is not just meant to reproduce existing methods, but to provide a foundation for future research. We hope that it will lower the barrier to entry and enable new discoveries that push the boundaries of physically based differentiable rendering.

ACKNOWLEDGMENTS

The authors thank David Hart and the NVIDIA OptiX team for tracking down a performance regression. Matt Pharr provided helpful feedback on an early draft. This work was supported by the Swiss National Science Foundation (SNSF) as part of grant 200021_184629.

REFERENCES

Sai Praveen Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased Warped-Area Sampling for Differentiable Rendering. *ACM Trans. Graph. (SIGGRAPH Asia)* 39, 6 (Nov. 2020).

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18 (2018).

Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. 1992. ADIFOR–generating derivative codes from Fortran programs. *Scientific Programming* 1, 1 (1992).

Benedikt Bitterli. 2016. *Rendering resources.* https://benedikt-bitterli.me/resources/.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

Ian Buck, T. Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph. (SIGGRAPH)* 23, 3 (2004).

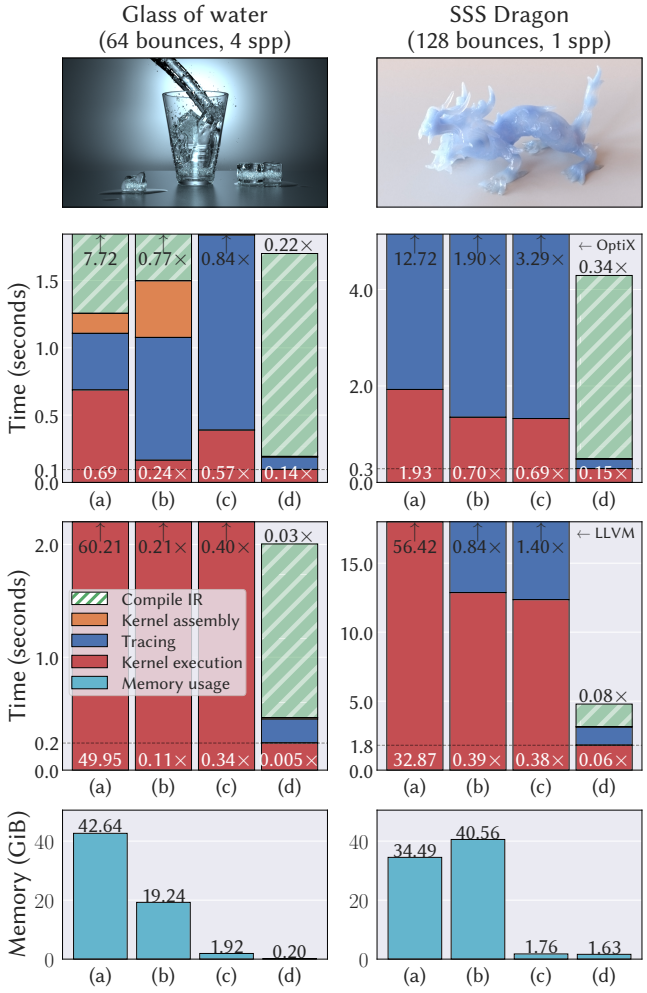


Fig. 16. Standard methods like path tracing can be differentiated within Dr.Jit but require checkpointing. This figure compares the behavior of such a memory-intensive approach to a specialized PBDR method. **(a)** Reverse-mode differentiation of a standard path tracer in wavefront mode. **(b)** Path tracer with optimized polymorphic calls. **(c)** Wavefront PRB with optimized polymorphic calls. **(d)** PRB compiled to a megakernel.

Table 1. Kernel sizes in thousands of IR operations averaged over the three benchmark scenes. This table provides numbers for primal rendering, reverse-mode (reparameterized) PRB, and the ratio relative to the primal column.

	Primal	PRB	(ratio)	Repa. PRB	(ratio)
wavefront	188.94	649.25	×3.44	6012.45	×31.82
+ vcall rec.	216.28	969.53	×4.48	6885.09	×31.83
+ vcall dedup.	141.21	654.49	×4.63	1042.37	×7.38
+ vcall opt.	138.74	495.65	×3.57	1103.82	×7.96
+ const prop.	127.13	448.29	×3.53	907.36	×7.14
+ value numb.	98.55	334.59	×3.39	800.07	×8.12
+ loop rec.	10.50	36.45	×3.47	78.77	×7.50
+ loop opt.	10.49	36.19	×3.45	78.69	×7.50

Brent Burley. 2012. Physically Based Shading at Disney. In *ACM SIGGRAPH Talks*.
Brent Burley. 2015. Physically Based Shading in Theory and Practice: Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering. In *SIGGRAPH Courses*.
Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. (2014). arXiv:1410.0759
Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP (July 2018).
Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Trans. Graph.* 37, 3 (2018).
Tess Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Trans. Graph. (SIGGRAPH)* 30, 4, Article 107 (July 2011), 12 pages.
Ioannis Gkioulekas, Anat Levin, and Todd Zickler. 2016. An evaluation of computational imaging techniques for heterogeneous inverse scattering. In *European Conference on Computer Vision (ECCV)*. Springer.
Google. 2017. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla (Accessed Jan 16, 2022).
Andreas Griewank et al. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989).
Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Vol. 105. SIAM.
Pat Hanrahan and Jim Lawson. 1990. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*.
Laurent Hascoet and Valérie Pascual. 2013. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)* 39, 3 (2013).
Yong He, Kayvon Fatahalian, and T. Foley. 2018. Slang: language mechanisms for extensible real-time shading systems. *ACM Trans. Graph. (SIGGRAPH)* 37, 4 (2018).
Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).
Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph. (SIGGRAPH Asia)* 38, 6 (Nov. 2019).
Michael Innes. 2019. Don't Unroll Adjoint: Differentiating SSA-Form Programs. arXiv:1810.07951
Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki. (Accessed: Jan 16, 2022).
James T. Kajiya. 1986. The Rendering Equation. *SIGGRAPH Comput. Graph.* 20, 4 (Aug. 1986).
Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*.
Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA.
João Rui Leal and Brad Bell. 2017. *CppAD 2017*. https://doi.org/10.5281/zenodo.836832
Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized production path tracing. In *Proceedings of High Performance Graphics*. ACM.
Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA (Nov. 2018).
Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018a. Differentiable Monte Carlo Ray Tracing Through Edge Sampling. *ACM Trans. Graph. (SIGGRAPH Asia)* 37, 6 (Dec. 2018).
Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018b. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (SIGGRAPH)* 37, 4 (2018).
Seppo Linnainmaa. 1976. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics* 16, 2 (1976).
Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing Discontinuous Integrands for Differentiable Rendering. *ACM Trans. Graph. (SIGGRAPH Asia)* 38, 6 (Nov. 2019).
William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph. (SIGGRAPH)* 22, 3 (July 2003).
Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. 2004. Shader Algebra. *ACM Trans. Graph. (SIGGRAPH)* 23, 3 (Aug. 2004).
Michael D McCool, Zheng Qin, and Tiberiu S Popa. 2002. Shader metaprogramming. In *Proceedings of the SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*.
William S Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. (2020). arXiv:2010.01709
William S Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In *Proceedings of SC21*.
Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. 2020. Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering. *ACM Trans. Graph. (SIGGRAPH)* 39, 4 (July 2020).
Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Trans. Graph. (SIGGRAPH Asia)* 38, 6 (Nov. 2019).
John F Nolan. 1953. *Analytical differentiation on a digital computer*. Ph.D. Dissertation. Massachusetts Institute of Technology.
Melissa E. O'Neill. 2014. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905. Harvey Mudd College, Claremont, CA.
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 Trans. Graph. (SIGGRAPH)* 29, 4 (July 2010).
Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP (Aug. 2021).
Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008).
Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2020. *Implementation of the forthcoming 4th edition of Physically Based Rendering: From Theory to Implementation*. https://github.com/mmp/pbrt-v4
Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE.
Lev Pontryagin. 1962. *Mathematical theory of optimal processes*. CRC Press.
Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: Generating Renderers without Writing a Generator. *ACM Trans. Graph. (SIGGRAPH)* 38, 4 (July 2019).
Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph. (SIGGRAPH)* 31, 4, Article 32 (July 2012).
Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP (July 2019).
Jeffrey Mark Siskind and Barak A Pearlmutter. 2018. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33, 4-6 (2018).
Bert Speelpenning. 1980. *Compiling fast partial derivatives of functions given by algorithms*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
Delio Vicini, Sébastien Speierer, and Wenzel Jakob. 2021. Path Replay Backpropagation: Differentiating Light Paths Using Constant Memory and Linear Time. *ACM Trans. Graph. (SIGGRAPH)* 40, 4 (Aug. 2021).
Yu M Volin and GM Ostrovskii. 1985. Automatic computation of derivatives with the use of the multilevel differentiating technique—1. algorithmic basis. *Computers & mathematics with applications* 11, 11 (1985).
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. (SIGGRAPH)* 33, 4 (July 2014).
Robert Edwin Wengert. 1964. A simple automatic derivative evaluation program. *Commun. ACM* 7, 8 (1964).
Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. 2021. Monte Carlo Estimators for Differential Light Transport. *ACM Trans. Graph. (SIGGRAPH)* 40, 4 (Aug. 2021).
Cheng Zhang, Bailey Miller, Kan Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-space differentiable rendering. *ACM Trans. Graph. (SIGGRAPH)* 39, 4 (2020).

A RELATIONSHIP TO ARRAY PROGRAMMING

This section presents microbenchmarks comparing Dr.Jit to a representative array programming framework, specifically JAX [Bradbury et al. 2018]. JAX is powerful a frontend to the XLA compiler [Google 2017], which analyzes the graph structure of the desired computation, *fusing* sequences operations as part of a heuristically guided clustering process. This involves tensorial optimizations

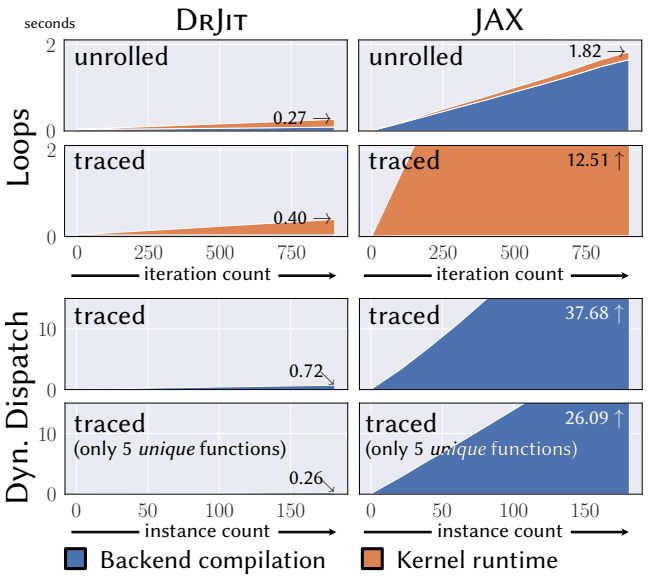


Fig. 17. Micro-benchmark comparing the tracing features of Dr.Jit and JAX. The top half benchmarks a simple loop with an update of the form `x=(x+1)∧x` (`x` is an 1D array of 10^9 32-bit integers) with loop counts increasing from 1 to 1000 on the horizontal axis and combined compilation and runtime in seconds on the vertical axis. The bottom half benchmarks dynamic dispatch to an increasingly large set of functions $f_1, f_2, \ldots$ implementing successively better approximations of the sine power series, i.e., $f_i = \sum_{k=0}^{i} -1^k/(2k+1)!\, x^{2k+1}$. In the last row, the functions f_i internally take `i` modulo 5, which means that there are only 5 unique functions. This in principle provides an opportunity to greatly reduce the size of the program.

like merging sequences of matrix-vector multiplications into matrix-matrix multiplications and selecting among the thousands of vendor-tuned kernels included in libraries like NVIDIA's cuDNN [Chetlur et al. 2014]. Steps that are not handled by an existing kernel require just-in-time compilation. JAX and XLA provide functionality to handle such custom computation including loop and dynamic function dispatch tracing resembling that of Dr.Jit. We initially implemented a small renderer using these primitives, but found that compilation timed out on nontrivial examples. Figure 17 benchmarks XLA tracing primitives to better understand these issues.

When unrolling loops, we observe significantly increased backend compilation time in JAX. Tracing loops via `jax.lax.fori_loop()` addresses this, but the generated code curiously commits loop state to memory at every iteration, which increases runtime cost. JAX also provides `jax.lax.switch()`, which produces a `switch{}`-like statement that effectively merges all functions into the body of the generated kernel. This produces a very large IR representation that exacerbates the cost of steps like register allocation. Dr.Jit instead generates indirect calls to subroutines that admit separate compilation. In the context of rendering, dynamic dispatch often targets functions that later turn out to be identical, which Dr.Jit exploits to reduce backend compilation time to a constant. XLA detects and exploits these redundancies as well, albeit with a runtime cost that grows with the size of the input program.

B DYNAMIC DISPATCH TO CLOSURES

Instance attributes present a subtle but important challenge that must be handled while tracing methods. Consider the following rudimentary implementation of a textured Phong BRDF:

```
class Phong(mitsuba.BSDF):
    def __init__(self, albedo: TensorXf, exponent: Float):
        self.albedo = albedo      # Bitmap data (TensorXf)
        self.exponent = exponent  # Specularity (scalar Float)

    def eval(self, si: SurfaceInteraction3f, wo: Vector3f):
        # Convert UV into texel coordinates and perform a lookup
        resolution = Vector2u(self.albedo.shape)
        pos = dr.min(Vector2u(si.uv * resolution), resolution - 1)
        albedo = self.albedo[pos.y, pos.x]
        # Evaluate reflected direction and BRDF terms
        r = Vector3f(-si.wi.x, -si.wi.y, si.wi.z)
        return albedo * dr.inv_pi + dr.dot(r, wo) ** self.exponent
```

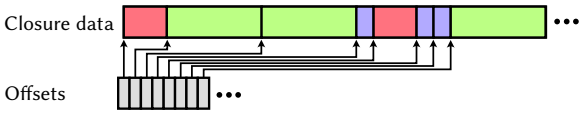
Evaluation accesses two attributes: `albedo`, which refers to a region in device memory, and a scalar `exponent` controlling specularity.

Tracing `eval()` must eventually produce a LLVM or PTX subroutine, whose signature is consistent with other implementations of this interface. It is logical that *explicit* function arguments like `si` and `wo` will be part of this interface, but how should *implicit* dependencies like `self.albedo` and `self.exponent` be handled? Each instance may reference different numbers and types of attributes, hence they cannot easily be turned into explicit parameters. It is also important to realize that "`self`" is purely a Python construct at this point that lacks a device-specific representation. A naïve solution would be to embed the raw attribute values in the generated IR:

```
def eval_impl_001(si_uv_x: float, si_uv_y: float, ...) -> float:
    albedo   = 0xfffe2f00 # Pointer to device memory
    exponent = 10         # Hardcoded constant
    # ... implementation (this would be LLVM/PTX IR in practice) ...
```

This is undesirable because it would break function deduplication, causing every `Phong` instance to produce a unique subroutine.

Instead of merely tracing functions, the system must handle *closures*, which refers to pairings of functions with data from a surrounding environment. Tracing and kernel assembly should then separate code from data so that optimizations remain effective. Dr.Jit transparently performs this optimization by detecting implicit variable dependencies while tracing function calls. Their contents are written into a contiguous array along with an auxiliary offset array that associates a closure data block with each instance.



Dr.Jit launches a builtin asynchronous kernel that collects dependencies from various locations in device memory to build this consolidated data structure. The total amount of device memory needed for them is small—usually on the order of 10–100 KiB per method call[4].

[4]Large device arrays like textures or volumes are accessed *indirectly*. The closure data array records a pointer to them rather than the data itself, while scalars are directly copied to avoid an extra indirection.