SLANG.D: Fast, Modular and Differentiable Shader Programming

SAI PRAVEEN BANGARU, MIT CSAIL and NVIDIA
LIFAN WU, NVIDIA
TZU-MAO LI, University of California San Diego
JACOB MUNKBERG, NVIDIA
GILBERT BERNSTEIN, University of Washington
JONATHAN RAGAN-KELLEY, MIT CSAIL
FRÉDO DURAND, MIT CSAIL
AARON LEFOHN, NVIDIA
YONG HE, NVIDIA

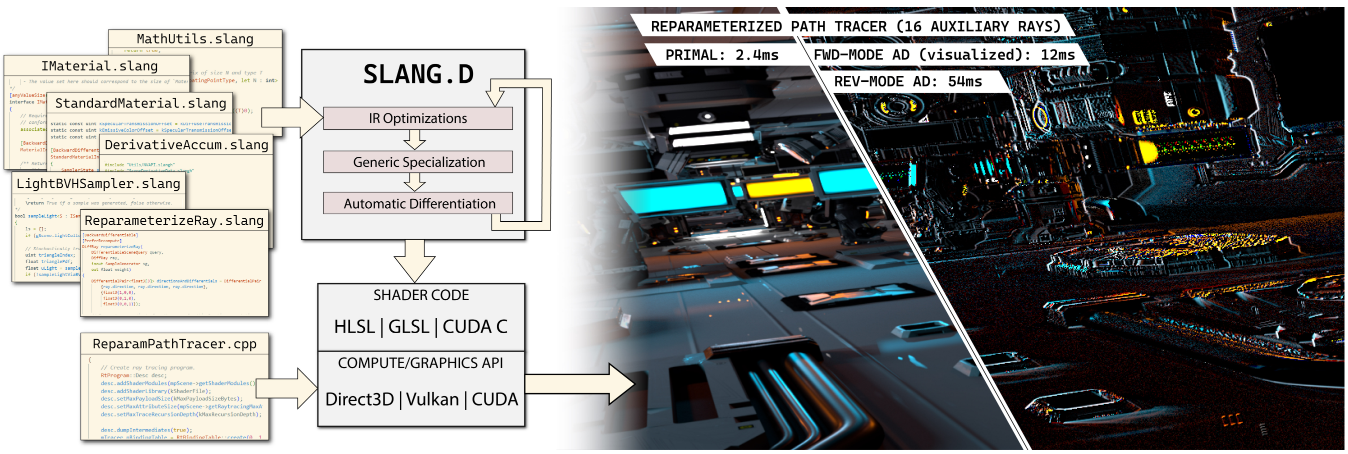


Fig. 1. SLANG.D is a shading language providing first-class automatic differentiation support that seamlessly interoperates with existing language features such as generics, dynamic dispatch, and complex control-flow. This figure shows the propagated derivatives on the Zero Day scene w.r.t camera's forward translation at 1920×1080, computed by a differentiable path tracer in the Falcor framework using the warped-area reparameterization [Bangaru et al. 2020] algorithm. SLANG.D allows reusing most of Falcor's existing components (material, lighting, and other utilities, which consist of 5,000 lines of shader code), by generating code that automatically propagates derivatives through these components. The generated derivative propagation code runs efficiently on an RTX 4090: the forward and reverse mode derivative propagation passes take 12ms and 54ms per sample. As a reference, the primal pass takes 2.4ms per sample.

We introduce SLANG.D, an extension to the Slang shading language that incorporates first-class automatic differentiation support. The new shading language allows us to transform a Direct3D-based path tracer to be fully differentiable with minor modifications to existing code. SLANG.D enables a shared ecosystem between machine learning frameworks and pre-existing graphics hardware API-based rendering systems, promoting the interchange of components and ideas across these two domains.

Our contributions include a differentiable type system designed to ensure type safety and semantic clarity in codebases that blend differentiable and non-differentiable code, language primitives that automatically generate both forward and reverse gradient propagation methods, and a compiler architecture that generates efficient derivative propagation shader code for graphics pipelines. Our compiler supports differentiating code that involves arbitrary control-flow, dynamic dispatch, generics and higher-order differentiation, while providing developers flexible control of checkpointing and gradient aggregation strategies for best performance. Our system allows us to differentiate an existing real-time path tracer, Falcor, with minimal change to its shader code. We show that the compiler-generated derivative kernels perform as efficiently as handwritten ones. In several benchmarks, the SLANG.D code achieves significant speedup when compared to prior automatic differentiation systems.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; **Rendering**; **Graphics systems and interfaces**.

Authors' addresses: Sai Praveen Bangaru, MIT CSAIL, Cambridge, MA , NVIDIA, Redmond, WA, sbangaru@mit.edu; Lifan Wu, NVIDIA, Redmond, WA, lifanw@nvidia.com; Tzu-Mao Li, University of California San Diego, San Diego, CA, tzli@ucsd.edu; Jacob Munkberg, NVIDIA, Lund, Sweden, jmunkberg@nvidia.com; Gilbert Bernstein, University of Washington, Seattle, WA, gilbo@cs.washington.edu; Jonathan Ragan-Kelley, MIT CSAIL, Cambridge, MA, jrk@mit.edu; Frédo Durand, MIT CSAIL, Cambridge, MA, fredo@mit.edu; Aaron Lefohn, NVIDIA, Redmond, WA, alefohn@nvidia.com; Yong He, NVIDIA, Santa Clara, CA, yhe@nvidia.com.

0730-0301/2023/12-ART262
https://doi.org/10.1145/3618353

Additional Key Words and Phrases: Automatic differentiation, differentiable graphics, differentiable rendering, shading language.

ACM Reference Format:
Sai Praveen Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Frédo Durand, Aaron Lefohn, and Yong He. 2023. SLANG.D: Fast, Modular and Differentiable Shader Programming. *ACM Trans. Graph.* 42, 6, Article 262 (December 2023), 28 pages. https://doi.org/10.1145/3618353

1 INTRODUCTION

Differentiable rendering pipelines have become increasingly important in solving computer vision and image synthesis problems [Hasselgren et al. 2022; Mildenhall et al. 2020; Zhang et al. 2022; Zhao et al. 2021]. Currently, to implement a differentiable renderer that runs on graphics hardware (GPU), a programmer faces two choices: 1) they can implement the renderer using a differentiable programming language (e.g., JAX [Bradbury et al. 2018] or Dr.JIT [Jakob et al. 2022]) and apply automatic differentiation [Griewank and Walther 2008], or 2) they can implement the renderer using traditional shading languages (e.g., HLSL), which are designed for working well with graphics hardware, and manually derive the derivatives. In this paper, we aim to bridge the gap between the two approaches by providing automatic differentiation for shading languages for high-performance differentiable rendering.

Designing programming languages for a GPU-based forward renderer already incorporates two main complexities. First, the requirement for having high-performance real-time rendering has led to graphics hardware that mixes both fixed-function and programmable components, such as vertex and fragment shaders in a rasterization pipeline, or intersection shaders in a ray tracing pipeline. Second, the shaders can consist of multiple subcomponents with shared code; for example, production materials can have hundreds of lines of code with shared parts between different materials. Moreover, when implementing a megakernel path tracer, the whole path tracer needs to be implemented in a single shader. Therefore, there is a strong demand for language features that make shaders both modular and efficient (abstract type systems, dynamic dispatch, generics, etc). Indeed, significant research effort has been invested in modular shading languages [Foley and Hanrahan 2011; He et al. 2018; Seitz Jr et al. 2019].

Designing automatic differentiation systems for shading languages further faces many unique challenges. Firstly, while shader programs in rendering are often embarassingly parallel, the backpropagation of them is not. For example, different pixels may need to accumulate differential quantities to the same texel during differentiation, causing race conditions. Secondly, GPUs have limited capability to dynamically allocate memory inside a thread for storing intermediate values during backpropagation. Thirdly, a renderer often contains components that are either not differentiable (e.g., an integer datatype), not required to be differentiated (e.g., the procedure for computing the probability of sampling a light [Zeltner et al. 2021]), or not directly differentiable and requiring special treatment (e.g., raycasting [Li et al. 2018a]). Finally, some differentiable rendering algorithms require nested or higher-order differentiation [Bangaru et al. 2020; Loubet et al. 2019].

These challenges make most existing automatic differentiation systems unsuitable for differentiating shaders. They are often not designed to run on a hardware graphics pipeline or exploit its fixed-function stages. Furthermore, they often lack the language features and type systems for writing efficient and modular shaders at the large scale of production code. We lay out a taxonomy of automatic differentiation systems and discuss it under the context of differentiating shader code in Sec. 2 and Appendix C.2.

In this work, we address these challenges by taking an existing, battle-tested shading language designed for high-performance and modular shader code, and augmenting it with first-class automatic differentiation support that can handle *all* features in the language, while making sure we preserve high performance. We choose to work with Slang [He et al. 2018] due to its first-class support for dynamic dispatch and generics that enable modular and fast code, and its compatibility with existing shading pipelines.

Following principles in hardware shading language design, an important design decision we make is to provide sufficient flexibility for users to write high-performance derivative code, while avoiding unpredictably complex program analysis. We provide language constructs for users to specify and control how they want to accumulate the derivatives, and to explore compute-memory trade-offs with checkpointing. Our system does not automatically make arbitrary backpropagation code race-condition-free. For example, while it is possible to implement an image convolution in SLANG.D, our system does not automatically recognize that the adjoint of the convolution is a correlation to generate race-free derivative code [Hückelheim and Hascoët 2022; Hückelheim et al. 2019; Li et al. 2018b]. Our system similarly will not automatically parallelize the assembly of a sparse Hessian matrix, or the matrix-vector product [Devito et al. 2017; Herholz et al. 2022; Schmidt et al. 2022]. However, the user can implement such optimizations themselves, assisted by our system.

Concretely, in this paper:

(1) We propose SLANG.D, a set of extensions to the Slang (Sec. 4) language. SLANG.D distinguishes differentiable and non-differentiable code by extending Slang's type system to handle differentiable types. SLANG.D provides primitives to instruct the compiler to initiate forward, reverse, and higher-order differentiation on Slang functions, as well as language mechanisms for user-defined derivative accumulation and checkpointing for preserving efficiency.
(2) We describe the design of the extended Slang compiler (Sec. 5) that computes derivatives of functions statically. Our compiler supports all features in Slang, including dynamic dispatch, specialization, control flow, and side effects. The compiler builds on the Slang compiler and can emit backend code for different stages of the shading pipeline.
(3) We evaluate our system by differentiating complex rendering systems and microbenchmarks (Sec. 6). In particular, we show that we can differentiate a complex and efficient path tracer in Falcor [Kallweit et al. 2022] by adding/modifying only 300 lines, while reusing 3,000 lines of existing material-related shader code and 252,000 lines of C++ host code. At the same time, we preserve the efficiency of Falcor. SLANG.D

also allows us to implement advanced differentiable rendering algorithms [Bangaru et al. 2020; Loubet et al. 2019] while reusing Falcor's code (Fig. 1).

2 RELATED WORK & BACKGROUND

We provide a brief taxonomy of automatic differentiation systems and where we position our system, and an introduction to the Slang shading language in this section. In Appendix C, we also provide additional background to automatic differentiation (C.1), a more complete discussion of automatic differentiation systems (C.2), and more background on shading languages (C.3).

2.1 Design Space of Automatic Differentiation Systems

Frontends. An automatic differentiation system can be a domain-specific language embedded in a host language with either *shallow embeddings* that avoid materialization of an intermediate representation (IR), or *deep embeddings* that materializes an IR using host-language constructs. Alternatively, it can be a standalone language with its own syntax. Earlier operator-overloading-based automatic differentiation systems are often shallow embeddings [Hogan 2014], recent embedded systems have moved towards deep embeddings to have more control over the IR construction [Bradbury et al. 2018; Jakob et al. 2022; Paszke et al. 2019]. Deep embedding languages often have to invent new syntax for describing loops and branches, as the control flows in the host language have different semantics than the control flows in the domain-specific language. Our system belongs to the class of stand-alone languages [Moses and Churavy 2020; Paszke et al. 2021]. This allows us to differentiate shader code with minimal changes. The challenge is that our system now needs to interact with all language features, including control flows, generics, and polymorphism, in Slang.

Execution model. Many automatic differentiation systems rely on *tracing*, where a host program is first run to produce an IR, and only later executed [Bradbury et al. 2018; Jakob et al. 2022; Paszke et al. 2019]. Tracing is convienent to implement in a deep embedding domain-specific language, however, it requires the programmer to reason about the two execution stages. In our case, since we directly build on the Slang language and compiler, we opt to not doing tracing. Thanks to Slang's generics language feature, our system still allows specialization to a known type or value to generate high performance code.

Intermediate representation. Automatic differentiation systems' IRs can differ in a few ways: whether they represent control flows, whether they contain type information for advanced language features such as dynamic dispatch, generics, and checking differentiability, and whether the IR is closed under differentiation for support of higher-order derivatives. Earlier tape-based systems often unroll the loops [Hogan 2014], whereas recent systems often incorporate loop representations (e.g., `tf.while_loop` and `dr.cuda.Loop`). Most automatic differentiation IRs lack type information and has limited support on language features. The ongoing Swift automatic differentiation work [Vytiniotis et al. 2019; Wei et al. 2021] and recent work on higher-order function differentiation [Huot et al. 2020; Krawiec et al. 2022]) addresses this. We incorporate the type system and language features in Slang into the IR, taking a similar approach to the Swift work, but applying it to shading languages. Many automatic differentiation systems do not support higher-order differentiation [Hu et al. 2020; Jakob et al. 2022], where our system ensures that the IR is closed under differentiation.

Program optimization. Automatic differentiation systems optimize their code based on the characteristic of the expected programs. For example, deep learning systems [Abadi et al. 2015; Paszke et al. 2019] optimize for the use case where each layer (convolution/attention) has high arithmetic intensity, which is usually not the case for shader programming. We follow the principle of shading language compiler design: we avoid complex program analysis for global optimization, while providing sufficient flexibility to the users to achieve high performance.

2.2 Slang Shading Language

Derived from HLSL, Slang is a shading language that supports modular development of complex rendering systems without compromising performance on GPUs [He et al. 2018]. As a result, Slang adopts many modern language features such as inheritance and interfaces. Slang is also a cross-platform language that targets Direct3D, Vulkan, CUDA and CPU. Slang has been adopted by many rendering products including Omniverse [Foley 2022], RTXRemix [NVIDIA 2023], Autodesk Aurora [Autodesk 2023], and Falcor [Kallweit et al. 2022].

The most important feature in Slang is the unification of shader specialization and dynamic dispatch with generics and interfaces. Slang's `interface` construct, also known as type classes or type traits in other languages, provides a natural way to express a type's capabilities. Listing 1 shows an exemplary `IMaterial` interface that all material implementations in a render must conform to.

```
interface IMaterial {
    float eval(float3 wi, float3 wo);
}
```

Listing 1. An Slang interface defines the requirements that all material implementations in a renderer must conform to.

This code defines that any type conforming to the `IMaterial` interface must provide a `eval` method for evaluating the material given an incoming and outgoing direction. With this interface, materials supported by the renderer can then be implemented as different types. Listing 2 shows two material implementations defined as `struct` types that inherits from the `IMaterial` interface.

The `IMaterial` interface allows developers to write code that uses the material subsystem, without exposing its implementation:

```
float3 evalLighting(
    IMaterial m, float3 L, float3 wi, float3 wo) {
    return L * m.eval(wi, wo);
}
```

The same code can also be written in a flavor that uses *generics*:

```
struct DiffuseMaterial : IMaterial {
    // ...
    float eval(float3 wi, float3 wo) { // ... }
}

struct RoughConductorMaterial : IMaterial {
    // ...
    float eval(float3 wi, float3 wo) { // ... }
}
```

Listing 2. Slang code showing different implementations of the `IMaterial` interface defined as struct types that inherit from the interface.

```
float3 evalLighting<M : IMaterial>(
    M m, float3 L, float3 wi, float3 wo) {
    return L * m.eval(wi, wo);
}
```

which can be invoked with the angle-bracket syntax: `evalLighting<DiffuseMaterial>`. In either flavor, if the Slang compiler can determine the `evalLighting` method is being invoked with a statically known material type, the compiler will specialize the method using the concrete type, enabling follow-up optimizations and potentially reducing the register pressure of the resulting code. If the type of `m` is only known at runtime, the compiler implements dynamic dispatch by replacing the `m.eval` call with a `switch` statement dispatching to a known material type based on the runtime type of `m`. Slang allows the developer to decouple how shader code gets written from the choice of how to tradeoff between specialization and dynamic dispatch.

We use the existing type system support for interfaces and generics to express the new logic for reasoning about differentiability. Additionally, by figuring out how the new automatic differentiation features interact with generics and interfaces, we can extend the dynamic dispatch support and other modularity features to differentiable code. Because Slang is compatible with most of existing HLSL code, the new automatic differentiation features can easily be adopted by both HLSL and Slang users.

3 OVERVIEW

Our goal is to design an automatic differentiation extension to Slang that satisfies the following desiderata:

- Given code written in Slang without differentiation in mind, our system should produce the derivatives of the code with **minimal changes of the forward code**. This allows us to inherit the language features in Slang and enable modularity.
- Our system should be able to distinguish between code that needs to be differentiated, and code that cannot or ought to not be differentiated. The system should report errors for invalid mixes of differentiable and non-differentiable code.
- Our system should have enough flexibility to allow users to achieve **high performance derivative code** that is at least as fast as highly-optimized manually differentiated code.
- Our system should support **higher-order differentiation**, whether for differentiating forward rendering methods that already require derivatives (e.g., ray differentials [Igehy 1999], or computing normals from neural signed distance fields), or for differentiable rendering algorithms that require higher-order derivatives [Bangaru et al. 2020; Loubet et al. 2019].
- Our system should **interact well with external code** that produces the scenes or processes the images (e.g., a convolutional neural network in PyTorch).

As a non-goal, our system will not perform complex program analysis to automatically achieve high performance. Instead, we provide sufficient flexibility for users to write fast code.

We achieve these goals by designing `SLANG.D`, a differentiable language and compiler that includes all of the existing features in Slang. This requires us to interact with Slang's type system and extend it to distinguish between differentiable and non-differentiable types, as well as handle higher-order differentiation. To achieve high performance and flexibility, we allow users to define custom derivatives for reverse-mode accumulation, which enables fast parallel reduction when multiple derivatives accumulate to the same location. We also allow users to annotate loops and functions to tell the compiler which checkpointing scheme to use.

Fig. 2 shows an overview of our system. The user code written with the new automatic differentiation features is parsed by the Slang compiler and checked with the extended type system (Sec. 4). Next, we implement automatic differentiation as an additional pass that is integrated into the code transformation loop in the Slang compiler backend (Sec. 5). The Slang compiler then generates target code (e.g., HLSL, GLSL or CUDA), and invokes the platform's downstream compiler to generate final executable code.

4 `SLANG.D` LANGUAGE DESIGN

In this section we describe the features added to the Slang language for automatic differentiation (AD). We also show how these features provide the `SLANG.D` compiler backend with the required information to perform automatic differentiation. In designing the language features, we borrowed many ideas from the ongoing work of adding automatic differentiation to Swift [Wei et al. 2021], and extended the ideas to support high-order differentiation in `SLANG.D`.

4.1 Differentiable Type System

One challenge of designing a differentiable language for interoperation with a traditional code-base is figuring out *which parts have to be differentiated*, and which parts can be left alone. This is especially non-trivial for differentiable rendering pipelines which use lots of components that are non-differentiable or have non-standard semantics for their derivatives. As an example, samplers used for estimating integrals are often not differentiated, or replaced with a different sampler entirely [Zeltner et al. 2021]. To make things more complicated, a single struct can have mixed values. In the example below, the *path payload* (`PathState`) carries differentiable information, such as throughput, but also undifferentiated information, such as path length.

```
struct PathState {
    float throughput; // requires grad
    uint  length;     // no grad
}
```

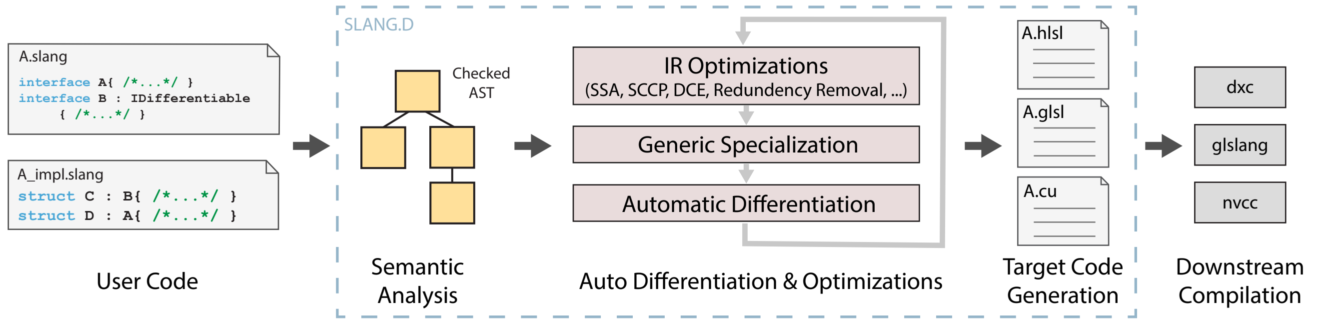


Fig. 2. An overview of the SLANG.D compilation pipeline. We extended Slang's front-end for parsing and checking differentiable SLANG.D code, and implemented automatic differentiation as an additional compilation pass that integrates into the backend's optimization loop. The Slang compiler then takes care of emitting target code and invoking downstream compiler for final executable code generation.

How do we infer which types are to be differentiated? What should the differential type be?

A popular approach in tracing-based automatic differentiation systems (e.g., Adept, PyTorch, and Dr. JIT) is to annotate differentiability per-*variable*, rather than per-*type*, by writing `var.req_grad=True`. Another approach, sometimes adopted by static compiled systems (e.g., Tapenade, Enzyme), is to use a fixed set of differentiable types (e.g., `double` is differentiable, `int` is not). This allows them to be more agnostic to language features and reuse automatic differentiation systems for different language frontends. Neither of these approaches makes it clear whether (e.g.) the output of a function ought to be differentiated. In the former case, code in an arbitrarily different part of the program may control whether or not differentiation happens (and require overriding via `detach`ing the output). In the latter case, a `double` output will be differentiated, regardless of the programmer's intent.

To facilitate type checking, we need to decouple **data** types (how a type is *stored*) with **semantic** types (what a type *means*). Two types can be stored the same way, yet have different semantics for differentiation. Slang's `interface` construct is a natural way to express such semantic information.

```
                                                          SLANG.D
interface IDifferentiable {
    associatedtype Differential: IDifferentiable,
        where Differential.Differential==Differential;
    Differential dzero();
    Differential dadd(Differential a, Differential b);
    Differential dmul<S:IScalar>(S s, Differential d);
}
```

Listing 3. The `IDifferentiable` interface defined in SLANG.D's standard library, specifying the requirements that a differentiable type must satisfy.

SLANG.D defines the `IDifferentiable` interface for objects that can be differentiated. Listing 3 shows the full definition of `IDifferentiable` in SLANG.D's standard library. Our definition uses the fact that a differential of any type is mathematically a *vector space*. This is a fundamental property that is a consequence of the linearity of differentiation, and holds no matter how complex the original computation may be. This definition is also equivalent to the `Differentiable` protocol in Swift.

The `IDifferentiable` interface defines requirements for the four mathematical properties of a vector space:

(1) `Differential`. It defines the data type to use to represent the differential. The `Differential` itself is required to be differentiable, enabling our goal of higher-order derivatives.
(2) `dzero()` *"Additive Identity"*. It defines the zero element for the vector space.
(3) `dadd()` *"Addition"*. It defines the sum of differentials, and is assumed to be commutative and associative.
(4) `dmul()` *"Scalar Multiplication"*. It defines the multiplication of a scalar with a differential, and is assumed to be distributive over `dadd()`.

```
                                                          SLANG.D
struct PathState : IDifferentiable {
    typealias Differential = float;
    float dzero() { return 0; }
    float dadd(float a, float b) { return a + b; }
    float dmul<S:IScalar>(S s, float d) { return s * (S)d; }

    float throughput;
    uint  length;
}
```

Listing 4. An example of a user-defined type that manually implements the `IDifferentiable` interface requirements.

Listing 4 shows an example of implementing the `IDifferentiable` requirements in a user defined type `PathState`, which uses a single `float` to carry the differential data.

By allowing the user code to define explicit type conformance to `IDifferentiable`, SLANG.D decouples the differentiability semantics from actual storage types. The flexibility offered by this decoupling can be useful in renderer code. For example, the Falcor framework represents a ray-geometry intersection using a *packed type* called `PackedHitInfo` where the primitive ID and barycentric coordinates are marshalled into a 64-bit integer to reduce bandwidth requirements. As such, it is incorrect to simply 'add' two packed data types. Thankfully, we can define the correct semantics for this type using our type system:

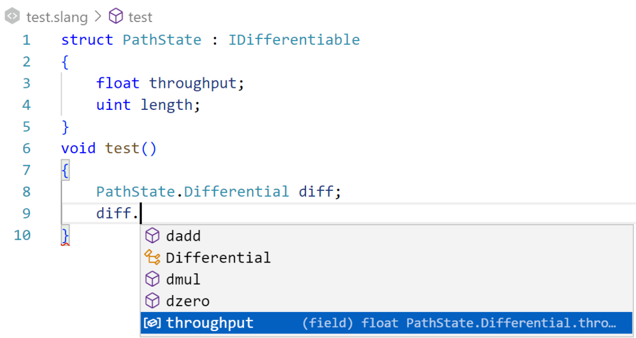


Fig. 3. Screenshot showing the SLANG.D plugin for Visual Studio Code listing the members available in a synthesized Differential type.

```
SLANG.D
struct PackedHitInfo : IDifferentiable {
    uint64 packed_primID_barycentricCoord;
    HitInfo unpack() { //... }
    typealias Differential = PackedHitInfo;
    Differential dadd(Differential a, Differential b)
        { return (a.unpack() + b.unpack()).pack(); }
}
```

Automatic Type Synthesis. While it is useful to be able to manually implement the `IDifferentiable` requirements for certain user-defined types, we often simply want to recursively differentiate all members of a `struct` type. Therefore, if a struct is marked as `IDifferentiable`, but no implementation is supplied, we automatically infer this default implementation. This is similar to Swift's `@memberwise` strategy.

To make it easier to work with types synthesized by the compiler in this way, the SLANG.D compiler synthesizes such `IDifferentiable` implementations during semantic analysis (i.e., type-checking), allowing for them to be surfaced through a code editor's language server plugin (a.k.a. intellisense). This enables developers to inspect synthesized types as they write code. Fig. 3 shows a screenshot of SLANG.D's Visual Studio Code plugin displaying the members of the synthesized `PathState.Differential` type in the previous example. Note that only the `throughput` field is included in the `Differential` type since the `length` field is not differentiable.

4.2 AD Operators as Higher-Order Functions

The second concern for differentiable language design is how to invoke derivative computation. Gradients of an expression can be computed either

(1) by invoking backpropagation from a variable (e.g., PyTorch's `x.backward()`): immediately propagate derivatives to all dependencies of the variable, or
(2) by invoking a higher-order function (e.g., JAX's `vjp(f)`): generate a new function that computes the reverse derivative.

Neither approach is outright superior in terms of expressiveness, but the variable-backpropagation approach is difficult to implement in our static compilation scheme.

In SLANG.D, derivative computation is expressed statically as higher-order function operations. Two built-in operators, `fwd_diff(f)` and `bwd_diff(f)` are used to call the *forward* or *reverse* derivative (Appendix C.1) propagation method of `f()` respectively. These operators can be viewed as higher-order functions that take a primal function and return a derivative function as a result.

While the body of a derivative function is created by the compiler (Sec. 5), we still need a convention for determining its type signature so the user can call the derivative function with the correct arguments. Consider a simple snippet: `z = x * y`. The forward-mode derivative of this snippet is the pair: `(z, dz) = (x * y, x * dy + dx * y)`, while the reverse-mode derivative would be `(dx, dy) = (dz * y, x * dz)`. For forward-mode, every input (x, y) needs to be paired with an additional differential input (dx, dy), while for reverse-mode, every input has a corresponding differential output, and the differential of the output (z) is now an input (dz).

To represent with such pairings, we first define a generic pair type (Sec. 4.2.1), we then incorporate it when deriving derivative function signatures (Sec. 4.2.2).

4.2.1 *Differential Pair Type.*

We provide a built-in generic type (`DifferentialPair`) to represent a primal and differential value pair and use it in both semantic checking and derivative code generation, defined as:

```
SLANG.D
struct DifferentialPair<T : IDifferentiable> :
      IDifferentiable {
    property T p;
    property T.Differential d;
    // Implementation of IDifferentiable requirements...
}
```

4.2.2 *Derivative Function Signatures.*

We then define the rules for deriving function signatures for both the forward and backward derivative functions. Given an original function that has type

```
func(T, U) -> R
```

where both R and T conforms to `IDifferentiable` and U does not conform to `IDifferentiable`, the forward derivative of `f`, `fwd_diff(f)`, will have type

```
func(DifferentialPair<T>, U) -> DifferentialPair<R>
```

The differentiable inputs and outputs are both paired with their differentials, and the non-differentiable parameters are left untouched.

On the other hand, the backward derivative `bwd_diff(f)`'s type will be:

```
func(inout DifferentialPair<T>, U, R.Differential) -> void
```

We use the `inout` modifier to allow the the primal of the pair as an input, and receive the differential back as an output. The derivative of the output of `f` is passed in as the final input parameter.

The complete signature transformation rules that covers differentiable and non-differential parameters that have `in`, `out` or `inout` directions are documented in Appendix E. These rules can sometimes be cumbersome to follow, but we alleviate this by including signature highlighting as a part of our Intellisense plugin.

4.2.3 Marking Differentiable Methods. To ensure runtime performance, SLANG.D requires the user to explicitly specify which functions should be differentiated so that the compiler does not produce unnecessary code that propagates derivatives through irrelevant functions. Users are expected to mark functions with the [Differentiable] attribute to make them available for differentiation. Attempting to call fwd_diff or bwd_diff on a method without this attribute will result in a compile-time error.

```
SLANG.D
[Differentiable]
float myPow(float x, int n) {
    float result = 1;
    for (int i = 0; i < n; i++)
        result *= x;
    return result;
}
```

Listing 5. A differentiable SLANG.D function that computes x^n.

Calling a non-differentiable function from a differentiable function is possible in SLANG.D. For example, Listing 5 shows a valid differentiable function that computes n-th power of x in SLANG.D. From the type system's point of view, the integer comparison i<n and increment i++ are treated as function calls into non-differentiable built-in intrinsic functions operator< and operator++. In this use case, the mixture of non-differentiable operations and differentiable operations presents no semantic ambiguity since the derivatives never flow through any non-differentiable part of the code.

However, allowing mixing differentiable and non-differentiable calls while requiring explicit Differentiable attributes on functions at the same time can lead to surprises and unexpected results. Consider this slightly modified version of myPow:

```
SLANG.D
// sqr is not marked as Differentiable and considered
// non-differentiable.
float sqr(float x) { return x*x; }

[Differentiable]
float myPow2(float x, int n) {
    float rs = 1;
    for (int i = 0; i < n; i++)
        rs *= sqr(x); // Error: derivative will be 0.
    return rs;
}
```

Since sqr is non-differentiable, its derivative is always zero. The user who wrote this code most likely wanted the derivative to propagate through the sqr function, but forgot to mark sqr as differentiable. Our type checker will report an error whenever it finds a call to a non-differentiable function in which some input arguments are IDifferentiable or the output is used where an IDifferentiable value is expected. If such a use of a non-differentiable function is intentional, the user can suppress this error by annotating with the no_diff keyword:

```
SLANG.D
result *= no_diff sqr(x); // Accepted by the compiler.
```

4.2.4 Differentiating through Interface Methods. To facilitate modularity, our higher-order functions fwd_diff(f) and back_diff(f) support abstract function calls in which the correct code to dispatch to cannot be statically resolved at compile time. As an example, consider the IMaterial interface definition in Listing 1, and its implementations in Listing 2. The member method eval of an abstract object could be any IMaterial implementation. So if we wanted to differentiate it, we would run into a problem:

```
IMaterial i = ... ;
// Cannot resolve i.eval statically
fwd_diff(i.eval)( ... );
```

In this example, i.eval could be either DiffuseMaterial.eval or RoughConductorMaterial.eval, or some other IMaterial implementation defined in a different, separately compiled module. The key to supporting this scenario is to ensure that an implementation of IMaterial.eval always has a derivative function defined. This is done by extending the [Differentiable] attribute to work on interface methods. When the compiler sees an interface method with the [Differentiable] attribute, it will add additional interface methods to represent the forward and backward derivatives of the [Differentiable] method. For example, given the following interface definition

```
SLANG.D
interface IMaterial {
    // All implementations must also be
    // marked with [Differentiable]
    [Differentiable] float eval(float3 wi, float3 wo);
}
```

the compiler will generate the full interface:

```
interface IMaterial {
    ...
    float eval(float3 wi, float3 wo);
    DifferentialPair<float> eval_fwd(
        DifferentialPair<float3> wi,
        DifferentialPair<float3> wo);
    void eval_bwd(inout DifferentialPair<float3> wi,
        inout DifferentialPair<float3> wo, float dOut);
}
```

With this interface, whenever the compiler needs to differentiate a call to the eval method dispatched from the IMaterial interface, it can simply emit a call to IMaterial.eval_fwd or IMaterial.eval_bwd without knowing which IMaterial is being called. For types that implements the IMaterial interface, the compiler will automatically derive the satisfying method for eval_fwd and eval_bwd from the user provided eval method.

4.2.5 Higher-order Application of AD. Note that the DifferentialPair type described above is itself marked differentiable, allowing a function generated by the AD pass to be differentiated again. A very important detail that makes higher order differentiation possible is the type *constraint* defined in Listing 3:

```
Differential.Differential=Differential
```

This constraint means that for any type D used as the Differential type of some type T, D's own Differential type must be D itself. This is needed to ensure that the automatic generation of `IDifferentiable` implementations can terminate without generating exponentially large amounts of code. We provide a detailed example illustrating this concern in Appendix. F. More importantly, this constraint is required for the `SLANG.D` compiler to synthesize `IDifferentiable` implementations for generic types. Consider the following type:

```
struct G<T:IDifferentiable> : IDifferentiable {
    T field;
}
```

In this case, an eager type system implementation that attempts to synthesize all `Differential` types for `G` may never terminate, because the generic `T` type does not provide any information on the length of the `Differential` type chain. Supporting this scenario would require a type system that is capable of handling infinite types. In contrast, by simply requiring that

```
T.Differential.Differential = T.Differential
```

the compiler can synthesize the differential type of `G` to be

```
struct G_Diff<T:IDifferentiable> : IDifferentiable {
    T.Differential field;
    typealias Differential = G_Diff;
}
```

Even if not mandated by the compiler, this constraint is almost always satisfied in practice. Generally, types in a differentiable program can be considered as a way to define the shape (dimension) of data to be differentiated. Once we have a representation of a differential value, differentiating it again will not change its shape, and therefore the second-order differential can be stored in the same type as the first-order differential.

4.3 Arbitrary User-Defined Derivatives

There are several situations where it is desirable to substitute a user-defined implementation in place of an automatically generated one. In differentiable rendering, there are two major reasons:

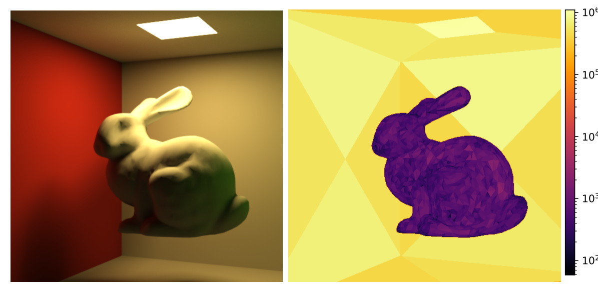
(1) *Intrinsic operations.* Primitives like `sin(x)`, and hardware-accelerated operations like interpolated texture sampling do not have a function body to differentiate, and need hand-coded derivatives or a reference implementation.
(2) *Parallelism and computational efficiency.* Shader programs often need to read from global data structures representing geometry, material parameters, lighting and camera setups, texture data, etc. When such a shader is backward differentiated, the resulting code needs to accumulate propagated derivatives into these global memory locations. For the sake of perforamnce, special care must be taken to minimize contention when implementing these accumulations.

Listing 6 shows an example of a load from a global variable holding material data (left), and an example of naïvely reversing the load to accumulate the reverse-mode derivatives (right). Since all threads access the same data, the simplest way to accumulate the derivatives is to use an atomic add. Unfortunately, if all threads

```
// Global data
MaterialData gMaterial;
// ...
float3 getAlbedo() {
    // Per-thread read
    return gMaterial.albedo;
}
```

```
// Global data
MaterialData.Differential
    d_gMaterial;
// ...
void bwd_getAlbedo(
    float3.Differential
        d_albedo) {
    // Naive reverse
    // accumulation
    atomicAdd(
        d_gMaterial.albedo,
        d_albedo);
}
```

Listing 6. An example shader code showing potential high write contention when accumulating the propagated derivative into a material parameter.



(a) (b)

Fig. 4. Non-uniform Access Patterns. (a) A Cornell box with a high-poly bunny rendered using a 1 bounce path tracer (b) The average number of times every face is accessed in a single iteration of a reparameterized differentiable path tracer [Bangaru et al. 2020] for computing geometry gradients. Note that the axis is in log-scale, and the larger triangles of the box are accessed orders of magnitude (1000x) more than the smaller triangles of the bunny mesh.

atomic write into the same memory location, this can result in a significant slow-down in runtime as writes from different threads get serialized by the hardware. Many solutions are available to alleviate this performance issue. We can perform the aggregation within a thread-group before accumulating to global memory. We can also allocate a hash grid for each derivative output, have each thread write to different locations based on a hash of the thread index, and then aggregate the elements from the hash grid in a follow-up pass.

However, there is no "one size fits all" solution to the write-contention problem. For example, Fig. 4 shows the order-of-magnitude difference in number of memory accesses across different parts of the scene during differentiable rendering. An optimal solution to lower the accumulation overhead is to use different aggregation strategies for different parts of the scene based on the access pattern, but this is not something our compiler can decide trivially.

We therefore enforce that global memory access instructions are non-differentiable by default, and prompt the user to implement access wrappers with user-defined code for derivative aggregation.

`SLANG.D` features `[ForwardDerivative]` and `[BackwardDerivative]` attributes for providing a custom forward or backward derivative propagation function. Listing 7 shows a custom backward derivative function for the `getAlbedo` function originally defined in

Listing 6. The custom backward derivative uses a wave-level reduction to aggregate derivatives across the current wave (thread-group) and then accumulate the aggregated derivative value once per wave — instead of having each thread perform its own global atomic write.

```
SLANG.D
MaterialData.Differential d_gMaterial; // Global data

[BackwardDerivative(bwd_getAlbedo)]
float3 getAlbedo() { // ... }

void bwd_getAlbedo(
    float3.Differential d_albedo) {
    var temp = WaveActiveSum(d_albedo);
    if (WaveIsFirstLane())
        atomicAdd(d_gMaterial.albedo, temp);
}
```

Listing 7. A custom backward derivative for `getAlbedo` in SLANG.D that efficiently accumulates the propagated derivative into global memory.

These decorations are applied to the primal method, and contain a reference to the derivative method. We also provide 'inverted' attributes that decorate the derivative method instead, useful for existing codebases (Appendix B.3)

Primal Substitutes for Intrinsics without Definitions. To allow propagating derivatives through hardware intrinsics (such as texture sampling) SLANG.D provides a *primal subsitution* mechanism. This allows users to provide a reference implementation for the intrinsic (e.g., a piece of SLANG.D code that performs tri-linear interpolation), while still allowing the compiler to differentiate the reference implementation and propagate derivatives.

This allows us to use high-performance intrinsics in the primal computations, without deriving the intrinsic operation by hand. Appendix D provides more details on how primal substitutes can be used to propagate derivatives through texture sampling operations.

Discussion of Alternatives. Many automatic differentiation systems provide custom derivatives, but how they inject the custom code back to the system are quite different. Deep learning systems (e.g., PyTorch/JAX) allow users to replace an operator's derivative with a custom one, which requires the users to keep track of all dependent variables of a function. Dr. JIT extends this idea to handle global variables in the context of the tracing. It does so by automatically tracking the dependent variables of a function, performing a closure conversion, and automatically accumulating the derivatives to the global buffers. However, neither systems allow users to specify how exactly they want to accumulate derivatives to the global buffers, such as the wave-level reduction scheme in Listing 7. Our approach is similar to other static AD systems (e.g., Enzyme), which simply replace the call with the provided custom function. Custom functions have all the flexibility of any other function in the source language. This avoids the need to track dependencies.

4.4 Checkpointing Primitives

Reverse-mode derivative propagation requires the primal computation results. Therefore the values computed during primal execution must be made available during backward propagation. There are primarily two methods: *Cache* the values during the primal pass or *Recompute* the expressions just before the primal value is used.[1]

When differentiating shader code with loops, the decision of which method to use is not trivial to make. Consider an example of a megakernel path tracer's main loop in a shader:

```
for (int i = 0; i < numBounces; ++i) {
    // ...
    Ray ray = sampleNextRay(... );
    // ...
    PackedHitInfo hitInfo = traceRay(... );
    ShadingData sd = computeShadingData(... );
    handleHit( ... );
    // ...
}
```

The generated adjoint loop will run the computation backwards, but in order to do so, it requires values generated in the primal loop. Here, `sd`, `hitInfo` and `ray` are all values that were generated in the primal loop, and required in the adjoint loop:

```
for (int i = numBounces - 1; i >= 0; ++i) {
    // ...
    bwd_handleHit( ... );
    // ...
    bwd_computeShadingData((sd, d_sd)... );
    bwd_traceRay((hitInfo, d_hitInfo)... );
    // ...
    bwd_sampleNextRay((ray, d_Ray), ... );
}
```

The simplest solution is to store all of these values into separate size-N arrays, but this easily overwhelms the small on-chip memory available to each GPU thread. Instead, a more efficient approach is to recompute some of the values (`sd`), and only store the light-weight values that are slow to recompute (`hitInfo`). However, automatically determining whether or not a value is slow to recompute can be difficult. Therefore, instead of attempting to guess the best answer, our compiler employs a light-weight heuristic and allows the user to control the behavior using attributes.

The heuristic defaults to recomputing everything, except across loop boundaries and function calls where recomputation *may* be expensive. For functions, we allow the user to specify a preference using a set of attributes: `[PreferCheckpoint]` and `[PreferRecompute]`. For loops, the user can unroll using `[ForceUnroll]` or split the loop index into two nested loops. The inner loop is unrolled, effectively reducing the size of the checkpoint allocation.

Using these strategies, the user can find a configuration that provides the best performance for their use-case: marking all the methods in the loop body with `[PreferRecompute]` will cause only the necessary loop state to be stored (the ray origin, direction and RNG state), which would work for large number of bounces. On the other hand, marking all methods with `[PreferCheckpoint]` will cause all loop state to be stored, avoiding recomputation for shorter loops whose checkpoints can fit into the L1 cache.

The $A\delta$ [Yang et al. 2022] compiler tackles the issue of register pressure using a custom Halide [2012] auto-scheduler to guide

[1]There is a third option of inverting the primal computation (e.g., [Vicini et al. 2021]). We leave a principled study of its use in general-purpose checkpointing as future work.

checkpointing decisions. While `SLANG.D` currently leaves this up to the user, the Slang reflection API allows external control over function attributes, and can be used to interface with a custom auto-scheduler if desired.

As a pure kernel language, `SLANG.D` is only concerned with *within-kernel* checkpointing, which is most impactful for megakernels that contain most of the program. Systems that launch multiple kernels, such as PyTorch and Dr. JIT, focus on *across-kernel* checkpointing [Kirisame et al. 2021], which is orthogonal to our work. Slang's API can be used by the across-kernel checkpointing system for querying the checkpoint layout of the kernel entry points.

We further discuss the backend implementation of checkpointing in Sec. 5.3.

5 SLANG.D COMPILER DESIGN

In the previous section, we discussed the user-facing language features to cleanly implement and maintain differentiable graphics components. The biggest challenge in supporting these language features is to make sure automatic differentiation works with all existing language features such as all forms of loops and arbitrary nesting of control flows and function calls, while providing sufficient user control on key decisions that affect runtime performance. We choose to implement our compiler without introducing any additional restrictions on user code to preserve the shading language's expressiveness for differentiable code. Our implementation is built upon the existing Slang compiler, which includes a full parser, an intermediate representation, optimization passes and code generation for a variety of targets.

In this section, we discuss the passes that had to be added to support automatic differentiation. Specifically, we first apply a preprocessing step that eliminates local variables to produce a single static assignment (SSA) intermediate representation (IR), and then brings the control flow graph to a reversible normal form. This normalized form is then differentiated through the linearization & transposition process [Radul et al. 2022], allowing both forward and reverse mode derivatives to be generated with the same process, and we apply a checkpointing pass to make primal values available during backward propagation even if the user code contains nested control flows and function calls. We use the `myPow` function in Listing 5 to illustrate the passes.

5.1 Preprocessing Pass

Our pre-processing steps are designed to bring the IR into a *normalized form*, by eliminating address and pointer types and bringing the control flow graph to a form that, upon inversion, can be expressed as a valid shader program. We found that these steps significantly simplify the design of the differentiation passes.

5.1.1 Address Aliasing Removal. The core logic to propagate derivatives backwards through an instruction is to accumulate the transposed derivative into the instruction's input values. If the instruction's input comes from a memory address that is updated several times by the user code, the generated propagation code must accumulate the derivative to the value at the memory address at the time the instruction was executed.

Fortunately, analyzing value identity at a memory address is easier in the Slang IR than in a general purpose programming language. This is because Slang does not allow pointers in user code, and global memory access are exposed through resource handles instead of pointers. Therefore, the only case where a pointer value can appear in the Slang IR is in accessing local variables or mutable function parameters. By definition, two different local variables can never alias. Additionally, since Slang inherits the copy-in, copy-out semantics of `inout` parameters [2] from HLSL, two mutable parameters in a function can never alias.

By leveraging these language characteristics, we can remove most pointers from the IR by applying the static single assignment (SSA) transform. However, the standard SSA pass in most compilers does not convert pointers of composite types (e.g., arrays and structs) into SSA values since in-place partial updates of large composite values cannot be expressed in the SSA form. For example, consider the following code that updates an array element:

```
A[i] = x
```

If we choose to turn `A` into an SSA value, the update operation will be represented as:

```
A1 = update A0, i, x
```

Which has the semantic of creating a copy of the array `A` with `i`-th element replaced with `x`. This can lead to performance issues if the copy is still needed when we come out of the SSA form.

In `SLANG.D`, the only situation where address aliasing may occur is when an array is being accessed with a dynamic index. While it may be possible to handle this directly in the AD passes, we choose to convert all local variables and parameters, including those of composite types into SSA values for implementation simplicity. The consequence is that the compiler must take additional care to avoid introducing unnecessary copies of large composite values when transforming out of the SSA form. To do so, the compiler tracks the variable or parameter from which each SSA value is constructed (in the above example, both `A0` and `A1` are originated from `A`), and coalesce the SSA values back to the same variable if permitted by the interference analysis after the AD passes.

5.1.2 Control Flow Normalization. Our decision to work in Slang results in a small complication due to its choice to emit code that is itself in a shading language (e.g., HLSL) rather than machine code (e.g., SASS). On the one hand, this takes advantage of powerful downstream optimizations, with the caveat that our generated derivative IR must be representable in the target language.

When it comes to control flow, shading languages only support *structured* primitives like `if-else` branches, `for`/`while` loops and `switch` statements. Arbitrary `goto` statements are not allowed in most shading languages including Slang since they can create situations where diverged control flow never re-converge, breaking the requirement assumed by many GPU architectures. Since reverse-mode derivatives require inverting the control flow, we must make

[2] When calling a function with an `inout` parameter in HLSL/Slang, a temporary variable is created and initialized to the value of the argument before entering the callee. The value of updated temporary variable after the call is written back to the argument.

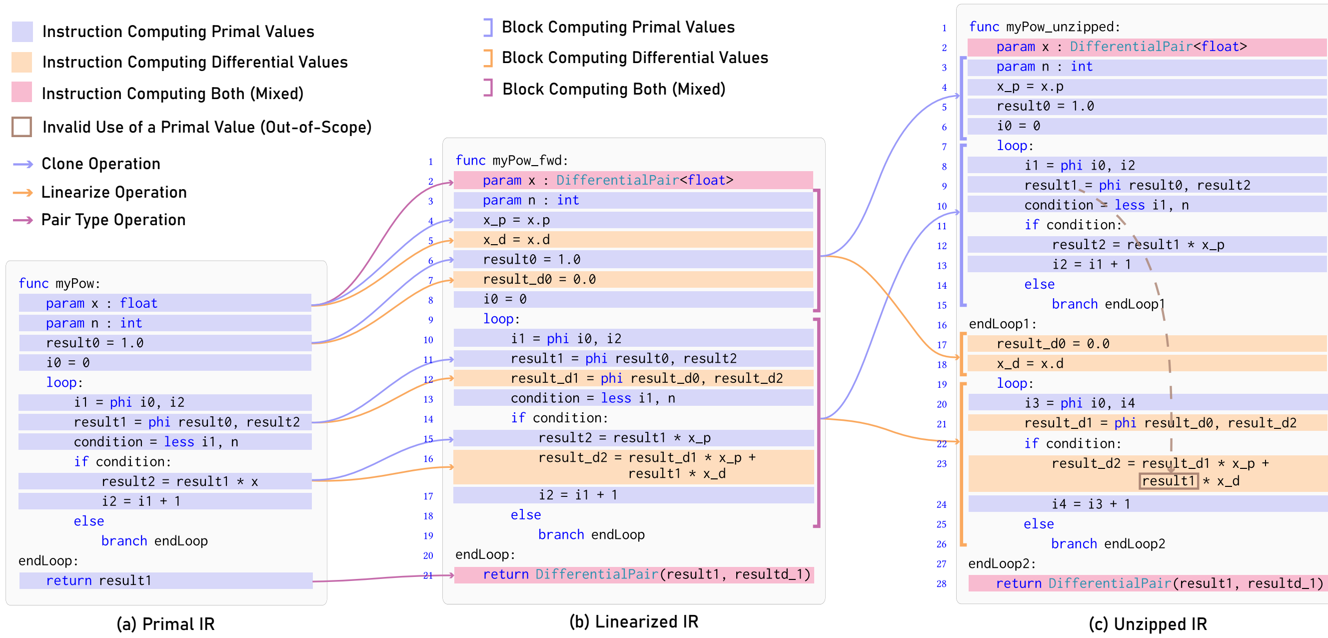


Fig. 5. Pseudo IR of the myPow function after the linearization and unzipping passes. **(a)** IR after the preprocessing pass. **(b)** IR after the linearization pass. The result is a forward derivative propagation function that computes both the primal result x^n and the propagated derivative result_d. **(c)** IR after the unzipping pass. The function is separated into two parts, where the first part contains only the primal computations, and the second part contains only the derivative propagation logic. All the blocks, including the loop, have been split into two to accommodate this transform. The result still propagates derivatives forwards. The IR in this form is no longer a valid program (e.g., `result1` is accessed out-of-scope), and will be legalized later during the checkpointing pass (Sec. 5.3).

sure that the resulting control flow is still representable with the supported control flow primitives. Instead of tackling the general problem of restructuring a reversed CFG into a structured form, the SLANG.D compiler *pre-processes* the CFG to bring it into a *trivially reversible* form, so that reversing can be done by simply flipping the direction of edges in the CFG without breaking the overall control flow structure. This is achieved using a control flow normalization pass, detailed in Appendix G, that eliminates non-trivial control instructions like `break`, `continue` & multiple `return` statements.

Fig. 5(a) shows the pseudo IR for `myPow` after the preprocessing steps. For clarity, we omit the detailed basic block structure and branching instructions in the listing and use the high level syntax instead to illustrate the control flow.

5.2 Automatic Differentiation Pass

With the input code normalized, the compiler can proceed to run a series of automatic differentiation passes to generate derivative propagation code. Our differentiation passes are inspired by the linearize-then-transpose idea [Radul et al. 2022], with modifications for our imperative-style IR. The idea is to generate forward derivative first (linearization), if the backward derivative is requested, we then further transform the forward derivative code (transposition). The AD passes are invoked 'on-demand' when the compiler encounters a call to `fwd_diff` or `bwd_diff`.

5.2.1 Linearization. The linearization step generates a forward-mode derivative function by differentiating each instruction of a differentiable type and inserting the derivative right after in the original function. The differentiable types are obtained from the front-end by checking which types inherit `IDifferentiable`.

Fig. 5(b) shows the generated code that propagates differentials of the input (`x.d`) to the output (`result.d`), while also computing the primal value itself (`result`). The linearization pass first transforms all differentiable parameters into `DifferentialPair` type so that the rest of the function can access both the primal value and derivative from the input parameter. Next, the pass emits new code right after each differentiable instruction that propagates the derivatives from the input operands to the original instruction's output. In this case, the pass emits new instructions to compute `result_d0`, `result_d1`, `result_d2`. This step operates locally on each differentiable instruction, and does not require modifying the control flow graph since both the primal and differential instructions flow in the same direction. See Radul et al.'s work [2022] for a fuller explanation of linearization rules for each type of instruction. If only forward derivative is requested, then the differentiation pass is done and we return the resulting function.

5.2.2 Transposition. If the user is requesting the backward derivative, the compiler must continue the transformation by transposing only the instructions that are computing differential values. The transposition pass is broken down into three steps. The compiler first *unzips* all blocks into primal & differential parts, transposes the instructions in each differential block, and reverse the control flow.

Unzipping. This step is responsible for re-ordering all primal instructions to before the first differential instruction. Since our IR contains control flows, unzipping involves duplicating the control flow graph. We create one copy of each block, and move all differential instructions to that block. Fig. 5(c) shows the `myPow` function after the unzipping pass.

Transposition within each block. In the second step, all instructions in each differential block have their order reversed, and the compiler transposes each derivative instruction in reverse order (last instruction in a block is processed first) to propagate derivatives backwards from the output differentials to the input differentials.

Due to the linearity of differentiation, there are only two fundamental rules for transposition: every multiplication `da = c * db` becomes a multiply-accumulate: `db += c * da`, and an addition `da = b + c` becomes two accumulations: `db += da; dc += da`. All other rules are derived from these two [Radul et al. 2022].

Reversal of control flow. Finally, the control flow of all the differential blocks are reversed, and since the IR is in a normalized form, the resulting graph is automatically valid. The only special case we take care of is inverting the loop index computation in for loops, for example `for(i=0; i<N; i++)` becomes `for(i=N-1; i>=0; i--)`. We also insert a counter variable for loops that do not have an induction variable.

Fig. 6(d) shows the `myPow` function after the transposition pass.

5.3 Checkpointing Pass

In the transposed code, the differential blocks can reference values generated in the primal blocks. For example, the instruction at Fig. 6(d), line 23 references `result1`, which is computed in the primal loop (line 7-15). This reference is invalid because `result1` is no longer available at line 23. In other words, the definition of `result1` does not dominate the use site at line 23. In this case we need to obtain the value of `result1` computed at the same loop iteration of the use site.

The *checkpointing* step legalizes these invalid references created during the transposition pass by making the primal values *available* for the differential instructions. As discussed in Sec. 4.4, we can either choose to **cache** or **recompute** such primal values. We do this in four steps: (1) classify values as 'recompute' or 'cache' using a heuristic and user input, (2) store cached values into a static data structure, (3) clone the recomputation logic into differential blocks and (4) extract the primal checkpointing and reverse-mode logic into separate functions.

5.3.1 Classification. The `SLANG.D` compiler implements an abstract *policy* system that is given a global view-point of the call-graph and transposed contents of every function in scope. The policy is then responsible for classifying all primal instructions into 'cache' or 'recompute' sets. We currently implement a greedy policy that recomputes whatever is possible except for function calls where we incorporate user input through the attributes specified in Sec. 4.4. Additionally, our compiler enforces caching by default for loop state variables regardless of the policy to avoid worsening the computational complexity of the resulting derivative code.

5.3.2 Caching. Instructions marked for caching are placed into a regular variable (or a fixed size array for values generated inside loops) in the primal block and loads from it just before the use site. Typically, we coalesce all such variables into a generated `struct` type that we refer to as the *intermediate context*, which we place on the thread-local memory by default.

Our static allocation approach is in contrast to most implementations of checkpointing that dynamically allocate a tape (e.g., `malloc`) and resize as necessary [Moses et al. 2021], which is extremely slow on GPUs. Crucially, dynamic allocation is disallowed by shading languages since it forms opaque barriers in compiler optimization. Our approach allows the downstream compiler to inline and optimize intermediate values, and determine the exact amount of register/L1 space required by each thread. The scheduler can then adjust occupancy of the hardware units to fit all memory requirements onto fast, low-latency on-chip memory (i.e., the L1 cache).

5.3.3 Recomputation. Instructions marked for recomputation are cloned into their appropriate position in the differential blocks. The cloning process is repeated for all operands of the instruction until all dependencies are available in the differential blocks. For operands that are control flow dependent (such as ϕ nodes), the necessary control flow and blocks are themselves cloned, *except* for loops. This is because cloning loops this way can create nested loops, turning an $O(N)$ loop into an $O(N^2)$ loop. This only occurs when recomputing loop state values, and is the reason why these are always cached.

Fig. 6(e) shows the IR code for `myPow` after the checkpointing pass. The pass detects that the derivative propagation logic uses `result1` and `i1` computed in the primal blocks. Since both of them are loop state values, the classification step chooses the caching strategy to make them available for the differential blocks.

To implement the caching strategy, the compiler generates the `myPow_Context` type (line 1) to hold the values of `result1` at each iteration of the loop. Following our static allocation approach, we allocated a static array of size `MAX_ITERS` to store the values of `result1` at each iteration of the loop. Users can specify `MAX_ITERS` by decorating the loops with a `[MaxIters(n)]` attribute. In fact, the front-end will enforce that every loop in a differentiable function must either be marked as `[ForceUnroll]`, to unroll the loop at compile-time, or have a `[MaxIters(n)]`[3] decoration that the compiler can use as the size for intermediate allocations. Since `i1` is used as loop counter, the compiler optimizes the storage by storing only the last value instead of a full array of `i1` at every iteration.

With the intermediate context type defined, the compiler inserts writes to the context after each value is computed in the primal blocks (line 14,15), and inserts reads from the context to replace the illegal references at line 25,30. After the checkpointing pass, Fig. 6(e) is a valid program that correctly propagates derivatives backwards.

5.3.4 Extraction. After the checkpointing pass, the differential instructions are no longer directly referencing any values computed by the primal instructions: a primal value is either recomputed in the differential blocks, or stored into a context. This allows the compiler to separate the checkpointed function into two functions: one that contains the primal code and stores values into the intermediate context, and one that consumes the intermediate context to

[3]Currently, if the loop exceeds the specified maximum at run-time, it can result in undefined behavior due to out-of-bounds memory accesses, although often the downstream compiler can detect and produce a warning on such cases.

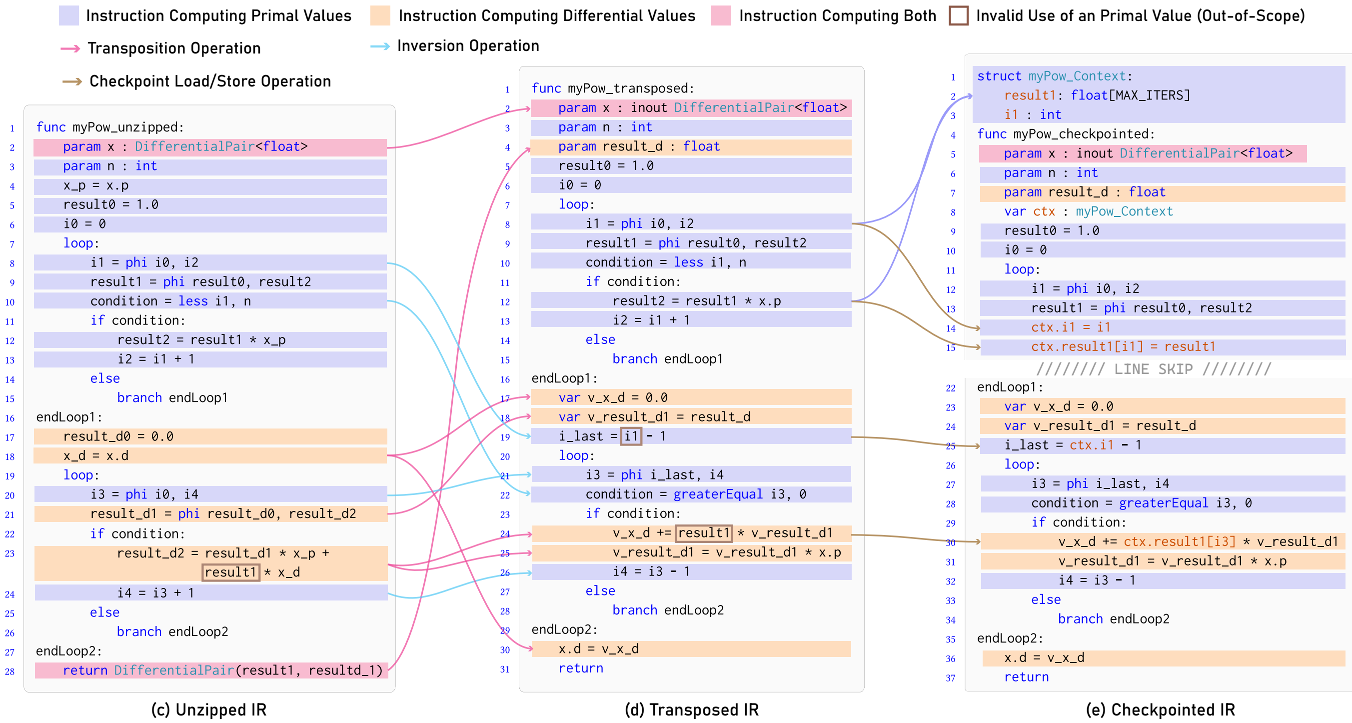


Fig. 6. Pseudo IR of the myPow function after transposition and checkpointing passes. **(c)** The IR after unzipping pass, duplicate of Fig. 5(c). **(d)** IR after the transposition pass. The instructions in differential blocks are reversed and transposed to accumulate propagated derivatives into their operands. Note the inversion of the loop index & condition to enable the loop to run backwards, and the allocation of intermediate variables to hold the accumulated derivatives. **(e)** IR after the checkpointing pass. An explicit context is created to hold the cached primal values, and context writes and reads are inserted into the program.

```
func myPow_makeCheckpoint:
    param x : float
    param n : int
    param ctx : out myPow_Context
    // ... (Fig. 6(e) Lines 5-15)

func myPow_bwd:
    param x : inout DifferentialPair<float>
    param n : int
    param result_d : float
    param ctx : myPow_Context
    // ... (Fig. 6(e) Lines 23-36)
```

Listing 8. Pseudo IR for the myPow function after the extraction pass. The checkpointed function is separated into two functions, one that computes the primal values and store them into the intermediate context, and one that uses the intermediate context to propagate derivative backwards.

propagate the derivatives backwards. This is done by *extracting* the primal code into its own function.

Listing 8 shows the transformed code of myPow after the extraction pass. The myPow_checkpointed function in Fig. 6(e) is split into two functions: myPow_makeCheckpoint, which contains all the primal code and the context writes to store the required primal values into the ctx output parameter, and myPow_bwd, which contains all the backward propagation code that reads the required primal values from the input ctx parameter.

The separation of checkpointed function into makeCheckpoint and bwd functions allows the caller of myPow to decide whether or not to cache or recompute the intermediate context of myPow. Consider a caller function mySqr that simply wraps a call to myPow:

```
float mySqr(float x) { return myPow(x, 2); }
```

Differentiating mySqr results in the following IR:

```
func mySqr_checkpointed:
    param x : inout DifferentialPair<float>
    param dOut : float
    var myPowCtx : myPow_Context
    myPow_makeCheckpoint(x.p, 2, out myPowCtx)
    myPow_bwd(inout x, 2, dOut, myPowCtx)
    return
```

In this code, the call to myPow_makeCheckpoint is a primal instruction that computes the primal value of myPowCtx, and the call to myPow_bwd is a differential instruction that consumes myPowCtx. The compiler can continue to make a decision on whether to cache or to recompute myPowCtx. As discussed in Sec. 4.4, we provide [PreferCheckpoint] and [PreferRecompute] decorations to control whether calls to a function should be recomputed or checkpointed. If myPow is marked as [PreferCheckpoint], then the compiler will transitively store myPowCtx in the intermediate context of mySqr, resulting the final make-checkpoint and backward propagation functions shown in Listing 9.

```
struct mySqr_Context:
    myPowCtx : myPow_Context

func mySqr_makeCheckpoint:
    param x : float
    param dOut : float
    param mySqrCtx : out mySqr_Context
    rs = myPow_makeCheckpoint(x.p, 2, out
           mySqrCtx.myPowCtx)
    return rs

func mySqr_bwd:
    param x : inout DifferentialPair<float>
    param dOut : float
    param mySqrCtx : mySqr_Context
    myPow_bwd(x, 2, mySqrCtx.myPowCtx)
    return
```

Listing 9. Pseudo IR for the mySqr function after the extraction pass, after applying [PreferCheckpoint] strategy for the call to myPow. The intermediate context for myPow is transitively included in the intermediate context of mySqr (mySqrCtx.myPowCtx).

5.4 Higher-Order Differentiation

By implementing automatic differentiation as static code transformation passes, the generated forward or backward propagation functions are no different from other user defined functions in IR. Therefore higher-order differentiation can be trivially supported by applying the AD passes repeatedly until we run out of fwd_diff or bwd_diff operations to process. For example, given the following function that initiates a higher-order differentiation:

```
void f(inout DifferentialPair<float> x) {
    bwd_diff(fwd_diff(mySqr))(x, 1.0);
}
```

After running the automatic differentiation passes once, the compiler will generate a function mySqr_fwd and f becomes:

```
void f(inout DifferentialPair<float> x) {
    bwd_diff(mySqr_fwd)(x, 1.0);
}
```

Since there is still a bwd_diff operation remaining, the compiler will run automatic differentiation one more time, and differentiate the previously generated mySqr_fwd function:

```
void f(inout DifferentialPair<float> x) {
    mySqr_fwd_bwd_Context ctx;
    mySqr_fwd_makeCheckpoint(x.p, 1.0, out ctx);
    mySqr_fwd_bwd(x, 1.0, ctx);
}
```

6 EVALUATION AND DISCUSSIONS

We evaluate whether SLANG.D achieves its goals (Sec. 3) using three case studies (Sec. 6.1) and two microbenchmarks (Sec. 6.2), while comparing to other systems that have been used for differentiable rendering before (Dr. JIT [Jakob et al. 2022] and Enzyme [Moses and Churavy 2020; Moses et al. 2021; Yu et al. 2022]). The case studies are larger applications of our system to differentiate complex rendering systems and implement involved differentiable rendering algorithms. The microbenchmarks are designed to show the effectiveness of writing derivative code in our system, showing that it achieves high-performance by providing sufficient flexibility to the user. At the time of writing, all SLANG.D extensions have been merged into the main Slang development branch and become a core Slang language feature. Performance numbers are evaluated using Slang release v2023.4.0, and on an NVIDIA RTX 4090 unless otherwise stated.

Ease-of-Use. Additionally, Appendices A & B lays out reasons why SLANG.D provides a better programming & debugging experience. We show that the single-instruction multiple-threads (SIMT) model is a much better fit for the fine control-flow of shader programs, unlike the N-dimensional-array (NDArray) model employed by alternative systems intended for specifying neural networks. We also elaborate on practical features like PyTorch interoperability (Appendix B.1), and debugging through print() intrinsics (Appendix B.2).

6.1 Case Studies

In Sec. 6.1.1, we first show that SLANG.D allows us to differentiate an entire path tracer and its material system in Falcor [Kallweit et al. 2022], with minimal change in code. We show that SLANG.D interacts well with the language features in Slang and scales well with the number of material instances.

Next, in Sec. 6.1.2 we show that we can concisely implement an advanced differentiable rendering algorithm named "Warped-Area Reparameterization" [Bangaru et al. 2020; Loubet et al. 2019] for addressing discontinuities in SLANG.D. The method is traditionally difficult to implement, partly due to its need for nested differentiation, which is not supported by existing systems.

Then, in Sec. 6.1.3, we use SLANG.D to replace the hand-written CUDA kernels in two complex inverse rendering pipelines that incorporate deferred-shading-based differentiable rasterizers [Hasselgren et al. 2021; Munkberg et al. 2022], showing that our system interacts well with external code. In all the case studies, our system is able to retain the high performance of the primal code, matching highly-optimized hand-written kernels.

6.1.1 Differentiating Falcor's Path Tracer and Material System. The Falcor real-time renderer, implemented in Slang and supports both Direct3D and Vulkan, provides a library of path tracers, materials, lights, samplers, and more by relying on Slang's generics and interfaces to abstract the complexity of each component. Building a differentiable renderer on top of Falcor means that we can leverage the state-of-the-art real-time rendering technology to speedup differentiable rendering, and vice versa.

We only need minimal modification to Falcor's code to make it differentiable (we address discontinuities Sec. 6.1.2). We make Falcor's material system differentiable by marking the material interfaces and implementations with [Differentiable] and using custom derivatives to provide optimized accumulation logic to propagate derivatives backwards into the material parameter buffer. We then invoke reverse-mode differentiation on the full renderer by

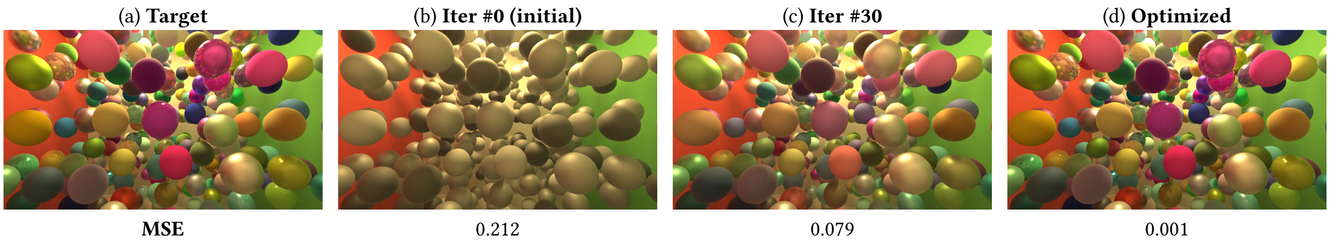


Fig. 7. Inverse-rendering result that optimizes thousands of material parameters simultaneously. This scene contains a 7^3 array of spheres and each of them has a unique material. At each iteration, our differentiable path tracer implemented with SLANG.D renders the scene at a resolution of 768×432, using 32 samples per pixel with 5 bounces.

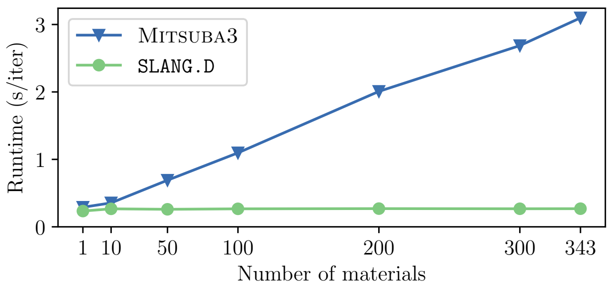


Fig. 8. We compare running times per iteration between our SLANG.D differentiable path tracer and Mitsuba3 on the material inverse-rendering optimization example shown in Fig. 7. As the number of materials increases, the running time of our implementation keeps constantly low because the differentiated SLANG.D shaders are compiled only once at the beginning of the optimization and can run very efficiently on the GPU. In constrast, Mitsuba3 has to re-trace and re-generate kernels at each iteration, leading to an increasing running time scaled with the number of materials.

calling `bwd_diff()` on the main rendering loop method. All modifications to the Falcor codebase are local, and most of the function implementations are kept unchanged.

Overall, we added and modified only 200 lines of shader code in Falcor's material system and an additional 100 lines for accumulating derivatives into global buffers, while reusing 3,000 lines of existing material-related shader code. Most of the existing host-side code (about 252,000 lines in C++) remains unmodified.

Our derivative code preserves the extreme high performance of Falcor[4], and interact well with the dynamic dispatch mechanism in Slang. We use the scene in Fig. 7 to test our system. The scene, depending on setting, contains one-to-three material types (a diffuse BRDF, a microfacet material [Cook and Torrance 1982], and Disney BSDF [Burley 2012, 2015]). Using the scene in Fig. 7, if we assign only a single material instance (Disney BSDF) to all objects in the scene, for an image resolution of 768×432 with 32 samples per pixel and 5 bounces, Falcor's primal rendering pass takes 58ms, while our reverse-mode pass takes 176ms. If we assign different material instances to all 343 objects in the scene (with all three material types), Falcor's primal rendering pass takes 68ms, while our reverse-mode pass takes 201ms. This is close to optimal in the compute-bound case: classical analysis [Griewank and Walther 2008] shows that the number of operations in reverse mode is bounded by $3-4\times$ of the primal pass. In practice, however, the memory traffic makes the bound unrealiable and complicates performance analysis.

We also compare our system's scalability with the number of material instances against Mitsuba3 (implementation based on Dr. JIT) using the same scene. We compare our system's performance with Mitsuba3's path replay backpropagation (`prb`) integrator [Vicini et al. 2021]. Fig. 8 shows the result. We note that it is not meaningful to directly compare the running times between Falcor and Mitsuba3 due to several implementation differences between the two systems (different underlying platforms, different scene representations, etc.)

The more important conclusion is that Falcor's performance remains mostly constant as the number of material instances increases, while Mitsuba3's running time grows linearly with the material instance count. This is because Dr. JIT needs to re-run the tracing and kernel generation steps at each iteration since the kernel computation logic is not known until the Python code that defines the optimization is executed. Dr. JIT's kernel caching does not help here because the kernel needs to be generated[5] (running the Python code that defines the computation) before the cache can be queried. This step can take up to 60% of total iteration time as the number of material instances increases. By contrast, with SLANG.D, Falcor always compile and cache a static shader during initialization & this shader works for any number of material instances.

The first half of Table 1 shows the total compilation time for the shaders used in primal rendering and the backward propagation pass. The compilation time is broken down into the time spent in the SLANG.D compiler and in the downstream DXC compiler. The compilation time increases from one material instance to 50 instances because in the one-instance case, only one material type is present in the shader, while the 50-instances case include code for all three material types. The shaders SLANG.D generated for 50-instances and 343-instances are exactly the same.

6.1.2 Warped-Area Reparameterized Path Tracer. We show that we can implement an advanced differentiable rendering algorithm using SLANG.D. The warped-area reparameterization (WAR) algorithm, proposed by Loubet et al. [2019] and extended by Bangaru et al. [2020], aims to address discontinuities in differentaible rendering. To differentiate an integral (commonly occurs in rendering) $\int_D f(\mathbf{x}, \theta)\mathrm{d}\mathbf{x}$ with respect to some parameter θ, where f can have discontinuities (e.g., visibility), the method applies a specific change-of-variable $\mathbf{x} = T(\mathbf{u}, \theta)$ to remove the discontinuities (see Appendix

[4]We build on the version from Clarberg et al. [2022], which is able to path trace dynamic scenes with billions of triangles at 1080p in real-time.

[5]This is reported in Dr. JIT's log output as `codegen_time`.

Table 1. Breakdown of our compilation time on the material inverse-rendering example (Fig. 7) and the warped-area reparameterization (WAR) example (Fig. 9). We separately measure the time for the SLANG.D compilation stage and the downstream (DXC) compilation stage. For the material example, we compare the compilation times when the scene contains different number of materials. For the WAR example, we compare the compilation times spent on the original undifferentiated shaders (primal) and automatic differentiated shaders using the forward-mode (fwd), the reverse-mode with the [PreferCheckpoint] strategy (rev, C), and the reverse-mode with the [PreferRecompute] strategy (rev, R). Times are measured on a machine with an AMD Ryzen 5950X CPU and 128GB DDR4 Memory.

Application	Section	SLANG.D	DXC	Total
Material (1 instance)	6.1.1	6.02 s	2.31 s	8.33 s
Material (50 instances)	6.1.1	8.23 s	4.55 s	12.78 s
Material (343 instances)	6.1.1	8.28 s	4.58 s	12.86 s
WAR path tracer (primal)	6.1.2	2.07 s	0.52 s	2.59 s
WAR path tracer (fwd)	6.1.2	2.29 s	2.10 s	4.39 s
WAR path tracer (rev, C)	6.1.2	3.61 s	9.81 s	13.42 s
WAR path tracer (rev, R)	6.1.2	3.45 s	8.92 s	12.37 s

B of Bangaru et al. [2020] for how T is defined):

$$\partial_\theta \int_D f(\mathbf{x}, \theta) d\mathbf{x} = \int_D \partial_\theta \left(f(T(\mathbf{u}, \theta)) \left(\det \left(\partial_\mathbf{u} T(\mathbf{u}, \theta) \right) \right) \right) d\mathbf{u}.$$

The equation above now has nested differentiation: we need to first differentiate the change of variable T, then differentiate the reparameterized integrand with scene parameter θ. Pseudo-code demonstrating this process is provided in Listing 10, with the complexity hidden behind the `reparameterize()` function. We are able to reuse 5,000 lines of existing shader code to build the WAR path tracer (we calculate the line numbers by summing up all falcor shader modules that are actually used in the example). The core WAR algorithm is implemented in 600 lines of SLANG.D code, and an additional 100 lines of custom derivative code is needed for accumulating derivatives into global buffers.

```
// Standard main loop
while (!path.is_terminated) {
   Ray ray =
        sampleNextRay(...);
   Intersection is =
        intersect(ray);
   //...
   path.radiance +=
        shade(ray, is);
   //...
}
```

```
// Reparameterized main loop
while (!path.is_terminated) {
   Ray ray =
        sampleNextRay(...);
   // Apply WAR
   ray = reparameterize(ray);
   Intersection is =
        intersect(ray);
   //...
   path.radiance +=
        shade(ray, is);
   //...
}
```

Listing 10. Reparameterizing an existing path tracer with WAR.

`reparameterize()` has a particularly complex implementation involving (i) tracing several additional rays (called *auxiliary* rays), (ii) computing the weighted mean of the projection of each intersection (called the *warp*), and (iii) computing the Jacobian determinant of this warp. Because of this complexity, existing systems such as Mitsuba3 and Redner [Li 2018], have chosen to implement it entirely by hand in the reverse-mode passes, making the implementation difficult to extend, modify and debug.

Since SLANG.D supports nested higher-order differentiation, our implementation comprises only the primal definition of the `warp` function, and uses the `fwd_diff(warp)` operator in the `reparameterize()` function to compute the divergence terms. The `reparameterize()` function is then automatically differentiated, effectively differentiating the `warp` function twice, eliminating the need for handwritten derivatives (Appendix H shows snippets of higher-order differentiation in action).

We found that WAR is also simpler to implement[6] in SLANG.D because we can use `fwd_diff` for debugging one parameter at a time, before switching to `bwd_diff` for optimizations, thus building confidence in the correctness of derivatives since both derivative functions are generated from the same primal code. On top of this, the reverse-mode code generated by SLANG.D performs efficiently, as shown in Table 2, which uses Mitsuba3's `direct_reparam` integrator[7] as a reference.

We also validate the generated differentiable renderer against finite differences and Mitsuba3's implementation, see Fig. 9.

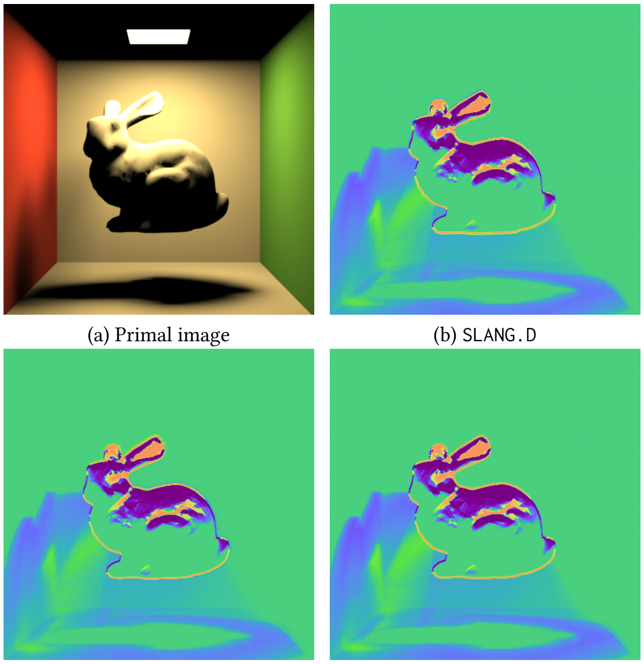
Discussion. Many optimization passes of our compiler work together to ensure that the differentiated code is efficient. The SSA data-flow analysis and our checkpointing policy determine that the loop tracing auxiliary rays does not have to store any state, allowing the reverse-mode loop to be very efficient. Further, Slang's arithmetic optimization and dead-code elimination passes automatically determine that `reparameterize()` has no effect in the forward pass, and eliminate the call to avoid tracing unnecessary rays.

Storing excessive state during backward propagation can lead to poor performance due to increased register pressure. We can achieve a significant speedup by marking everything within the main path-tracing loop as 'recompute', effectively causing SLANG.D to only store the minimum set of loop state variables and intersection results (124 bytes per ray, per bounce), and recomputing everything else during the reverse pass. Table 2 shows the running time differences when most of the intermediate terms are being stored (SLANG.D(C)) or recomputed (SLANG.D(R)). We also provided the runtime performance of Mitsuba3 as a reference.

6.1.3 Replacing Hand-Coded CUDA Kernels in Differentiable Rasterization Pipelines. Many differentiable rendering pipelines need to interact with machine learning frameworks such as PyTorch. For example, some methods rely on differentiable meshing procedures to convert scene representations [Hasselgren et al. 2022], and some methods need to process the rendered images with deep learning architectures [Liu et al. 2018]. However, PyTorch is not suitable for implementing high-performance rendering code. As a result, practitioners usually implement hand-coded CUDA kernels with manually-derived derivatives and wrap them as PyTorch operators. Here, we show that our system allows us to replace those hand-coded CUDA kernels and derivatives using SLANG.D code, and our compiler generates code that is equally efficient to highly-optimized

[6]Code available in the supplementary as "ReparameterizeRay.slang"

[7]We stress that Mitsuba3 and Falcor use different libraries and APIs. While we took precautions to match our implementation with Mitsuba3's, these numbers should be treated as a reference point and not as an exhaustive comparison.



(a) Primal image (b) SLANG.D

(c) MITSUBA3 (d) Finite differences

Fig. 9. Scene derivative with respect to the bunny's translation in the y-axis. (a) Primal image rendered with Falcor's path tracer written in Slang. (b) We extend the path tracer by implementing the warped-area reparameterization algorithm with SLANG.D and generate the derivative image using the forward-mode automatic differentiation. Our derivative visualization matches references generated by (c) MITSUBA3 and (d) finite differences.

Table 2. Performance measurements for the warped-area reparameterization results in Fig. 9. The primal and derivative images have a resolution of 1024^2, rendered with 1024 samples per pixel. We measure the wall-clock time used for running primal rendering, the forward-mode, and the reverse-mode automatic differentiation, respectively. Our implementation using SLANG.D with the [PreferRecompute] strategy, i.e., SLANG.D(R), is more efficient than the reference in MITSUBA3 and the SLANG.D(C) variant using the [PreferCheckpoint] strategy.

	MITSUBA3	SLANG.D(C)	SLANG.D(R)
Primal	1.16 s	0.54 s	0.54 s
Forward-mode	11.04 s	2.82 s	2.82 s
Reverse-mode	23.11 s	79.85 s	9.33 s

hand-written derivative code. In Appendix A, we further discuss our PyTorch interoperation that enables the drop-in replacement.

We first adapt the NVDIFFMODELING inverse rendering system, that employs a deferred shading based differentiable rasterizer [Laine et al. 2020] with handwritten CUDA kernels[8] for differentiable physically-based shading. We rewrite the handwritten CUDA kernels using both SLANG.D and EnzymeCUDA [Moses et al. 2021].

Table 3 compares our performance with PyTorch, manually-written CUDA derivatives, and EnzymeCUDA. Our system and EnzymeCUDA

[8]Code is available here: https://github.com/NVlabs/nvdiffmodeling

Table 3. Performance measurements for the differentiable physically-based shading kernel from NVDIFFMODELING [Hasselgren et al. 2021], measured on a launch size of [8,1024,1024] on a V100 GPU. The reported numbers are averages over 1000 iterations.

	PyTorch	Cuda	EnzymeCuda	SLANG.D
Primal pass	15.10 ms	1.07 ms	1.09 ms	1.18 ms
Primal+Bwd pass	55.64 ms	6.40 ms	6.83 ms	6.18 ms

are equally efficient to the manually-written CUDA code, while being much faster than PyTorch due to less memory traffic.

Next, we ported all custom CUDA kernels[9] from NVDIFFREC, a larger inverse rendering pipeline for joint shape, material, and lighting optimization [Munkberg et al. 2022].

The kernels perform loss computation (log-sRGB mapping and warp-wide reduction), tangent space normal mapping, vertex transform (multiplication of a vertex array with a batch of 4×4 matrices), and cube map pre-filtering (for the split-sum shading model). SLANG.D achieves the same performance as the handwritten CUDA code (Table 4), and reduces the number of lines of code by approximately 3×. We did not succeed in compiling these kernels with EnzymeCuda, due to the lack of support for warp-wide intrinsics.

Table 4. Performance measurements for all custom CUDA kernels from NVDIFFREC [Munkberg et al. 2022]. We measure the time of the primal and backward pass on a launch size of [8,1024,1024] on an A6000 GPU. The reported numbers are averages over 1000 iterations. The cube map pre-integration kernels do not have equivalent PyTorch versions, and were measured on cube maps of size [6,64,64].

Kernel	PyTorch	Cuda	SLANG.D
Loss	18.26 ms	2.46 ms	2.50 ms
Transform	6.19 ms	1.93 ms	1.92 ms
Normal	46.56 ms	6.04 ms	6.04 ms
Cubemap diffuse	-	86.38 ms	86.16 ms
Cubemap specular	-	21.99 ms	22.01 ms

6.2 Micro-Benchmarks

We design micro-benchmarks to test two of our core features: custom reverse-mode accumulation (Sec. 4.3) and checkpointing primitives (Secs. 4.4 and 5.3). We show that they achieve the desired high performance by providing sufficient flexibility to the programmer. The flexibility allows our system to produce code that is significantly faster than prior systems, and also differentiate code that was not allowed previously.

6.2.1 Long Loops: Effect of Checkpointing. We test the effect of checkpointing on performance using the following loop that is partially unrolled:

[9]Code is available here: https://github.com/NVlabs/nvdiffrec

```
// This is a Taylor series approximation of sin(x)
float sum = 0;
float term = x;
for (int i = 1; i < N; i+=UNROLL_AMT)
  [ForceUnroll]
  for (int j = i; j < i + UNROLL_AMT; j++) {
    sum += term;
    term *= -x * x * (1.0 / (2 * j) * (2 * j + 1));
  }
return sum;
```

The reverse-mode loop must remember/recompute the value of `term` at each iteration.

Since the inner loop is unrolled, only the outer loop's state needs to be checkpointed and the number of outer loop iterations can be controlled by changing `UNROLL_AMT`, which we treat as a shader specialization parameter. The unrolled instructions of the inner loop get *recomputed* at each iteration, adding redundant computation, but reducing the number of reads and writes that could potentially spill to global memory.

By changing the unroll amount, we can control the tradeoff between memory accesses and computations. Fig. 10 shows the effect. By picking an optimal unrolling factor (of 32), we see an order of magnitude speedup over checkpointing every iteration. Similar result is likely to show up in more complex loops such as the ones in differentiable renderers.

We implemented the same loop in Dr. JIT and EnzymeCUDA. Table 5 shows the performance comparison. Unfortunately, we ran into issues in both prior systems with this loop benchmark. Dr. JIT cannot differentiate *recorded loops* where the derivatives are propagating through a loop state variable that is being updated in each iteration (`sum` and `term`). While EnzymeCUDA is able to compile the loop, the generated derivative kernel crashes on larger iteration counts. A close inspection revealed that the crashes occurs when the iteration count exceeds the threshold upon which LLVM no longer unrolls the loop, suggesting potential issues related to memory allocations for the checkpointing state.

In contrast, SLANG.D's static-allocation checkpointing unloads the memory management to the downstream compiler, enabling supports for loops without the need for unrolling. The resulting code scales well to both large and small workloads. Since this kind of loop is not supported by Dr. JIT, we only include the performance numbers for a fully unrolled version of the program that assumed a statically-known loop count. The fully unrolled version is still slightly slower than the unrolling version written in SLANG.D. We attribute this to Dr. JIT launching additional kernels for derivative aggregation, and SLANG.D handles derivative computation and aggregation within the same kernel.

6.2.2 Shapes-2D: Derivative Aggregation & Dynamic Dispatch. Our second micro-benchmark demonstrates the performance impact of different strategies to accumulate derivatives into input parameters (Sec. 4.3), showing the value of giving user the control of this process. The Shapes-2D benchmark is a simple kernel that renders many 2D shapes using signed distance fields. The kernel is launched with one

[10] We used `#pragma unroll` for unrolling loops, but it appears that LLVM does not completely unroll the loop, causing Enzyme to crash on higher iteration counts

Table 5. Long Loops Micro-Benchmark. Performance comparison of reverse-mode AD on the sine-approximation loop for 10^9 elements, measured on an RTX 4090. We has to use statically known iteration counts for both Dr.JIT and Enzyme since the former cannot differentiate loop state, and the latter crashed when we used a dynamic iteration count, due to its heap-based checkpointing approach. On static loop counts, SLANG.D is faster than Dr.JIT because our method can handle gradient computation and aggregation within the same kernel, while Dr.JIT launches additional kernels for the latter. We also show how SLANG.D's dynamic versions compare here, with partial-16 unrolling performing the best. We conclude that SLANG.D is robust, scales proportional to the workload, and specializes well with statically known constants.

Iteration Count →	16	32	64	128
Static Iter Count				
Dr.JIT (unroll)	18.8ms	18.8ms	26.4ms	52.8ms
EnzymeCUDA (unroll[10])	39.7ms	74.3ms	CRASH	CRASH
EnzymeCUDA (static)	137.2ms	280.8ms	CRASH	CRASH
SLANG.D (unroll)	7.2ms	7.6ms	9.9ms	22.2ms
Dynamic Iter Count				
SLANG.D (partial-16)	69.2ms	72.2ms	81.5ms	137.0ms
SLANG.D (**no**-unroll)	3859.1ms	4117.8ms	4376.5ms	4956.1ms

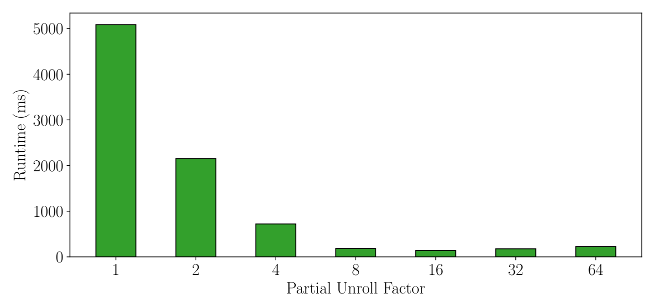


Fig. 10. The runtimes of the reverse-mode kernel generated by SLANG.D for the long-loop example, running the loop for 128 iterations on 10^9 elements of x. This plot illustrates the effect of checkpoint size, and consequently the register/memory pressure on the execution times of the simple long-loop benchmark. Checkpoint size is changed through partially unrolling the loop every $U = 1, 2, 4, 8, 16, 32,$ and 64 iterations. Since the outer loop only runs $\frac{N}{U}$ times, the checkpoint size also decreases proportionally. We observe that in this benchmark example, the execution time sharply decreases with the checkpoint size and hits diminishing returns at around $U = 32$.

thread per pixel, where each thread loops over a list of two different types of shapes (squares and circles), accumulating a total color for the pixel through additive blending. For simplicity, we did not implement any culling algorithms and the program has an $O(NM)$ complexity where N is the number of pixels and M is the number of shapes. We measure the runtime performance of a reverse-mode kernel that computes the derivative of output color with regard to all the input shape parameters (position, size and color). This benchmark also tests the performance of differentiation through dynamically dispatched methods, by defining an `IShape` interface and provide the signed distance field function for different shapes as separate types that implement this interface.

Since all pixels in the primal program reads from the same set of shape parameters at each loop iteration, the reverse-mode kernel can suffer from severe write contention when accumulating the derivatives. We compare the performance of three different derivative aggregation approaches: NAIVE atomic add, which simply performs a global `atomicAdd` whenever the derivative is propagated into a shape parameter and thus has the most contention, HASHGRID-κ which reduces contention by writing into a gradient buffer that is κ times larger, randomly hashing the pixel ID to pick an offset into the buffer. This buffer is then manually aggregated in a follow-up kernel. Finally, since custom derivative functions can be *any* shader code, the WAVESUM strategy uses warp-reduce intrinsics (e.g., `WaveActiveSum` in HLSL) to perform a single write per warp. WaveSum vastly reduces contention, but can only be applied if all threads in the warp are accumulating gradients of the same parameter. The assumption is true in this benchmark but not so in general programs and therefore cannot be assumed in a one-size-fits-all solution.

Our results in Fig. 11 show that NAIVE aggregation is an order of magnitude slower than WAVESUM, highlighting the importance of control over aggregation methods to combat contention. Fig. 11 also compares SLANG.D's performance using WAVESUM strategy against DR. JIT's fixed strategy of launching an additional atomic-scatter-add kernel to handle the aggregation. Because the SLANG.D code is able to take advantage of the special execution pattern in this benchmark, we are able to achieve an over 6x speedup over DR. JIT.

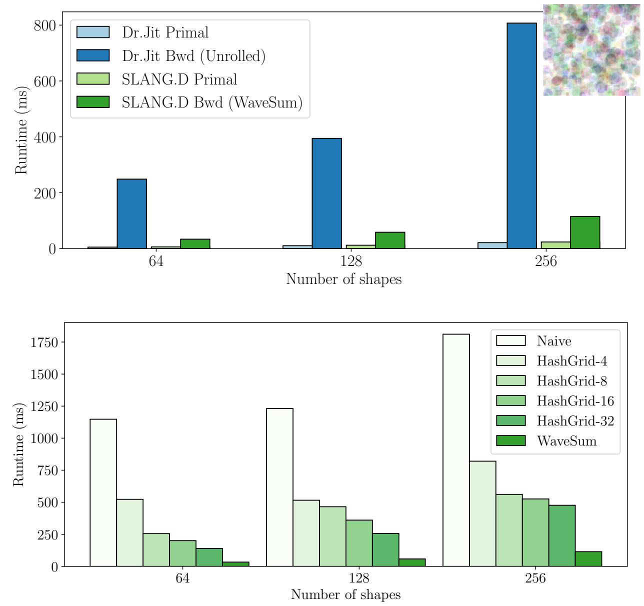


Fig. 11. SHAPES-2D DYNAMIC DISPATCH MICRO-BENCHMARK (**Top**) Performance comparison of the primal and backward derivative propagation kernels for the SHAPES-2D benchmark. The performance of the SLANG.D kernel that uses the fastest custom aggregation method (WAVESUM) is compared against the equivalent code written in DR. JIT. Both compilers were set up to target CUDA. The 4096×4096 image rendered by primal code is shown in the top-left corner. Two types of shapes are implemented: circles and rectangles. Each circle has 6 parameters (2 for position, 1 for radius, and 3 for color), and each rectangle has 7 parameters (4 for position and size and 3 for color.) The SLANG.D kernel is faster, and has lower overhead over the primal pass. This gap can likely be attributed to DR. JIT's use of function pointers for dynamic dispatch (instead of control flow), and a fixed derivative aggregation method (atomic-scatter-add). (**Bottom**) Ablation study of different aggregation methods. The order of magnitude difference between the best (WAVESUM) and worst (NAIVE) approach highlights the effect of aggregation methods on performance.

7 LIMITATIONS

Across-kernel differentiation. Being derived from the Slang shading language, SLANG.D inherits its limitations in terms of application scope and expressiveness. SLANG.D is intended for authoring differentiable shaders or compute kernels, and cannot differentiate programs that span multiple render passes or kernel launches. By contrast, systems like Dr. JIT and PyTorch that can generate multiple kernel launches from a single user function. This is because shader invocation is external to shader code, and is defined by the host code, driver and API. SLANG.D can be used to differentiate each stage, and write differentiable versions of the fixed-function units (e.g., rasterization), but the host code is responsible for invoking them in the right order & allocating the intermediate buffers.

Sub-optimal handling of local arrays. The SLANG.D compiler's implementation may currently generate suboptimal derivative accumulation code when the user code is updating a local array iteratively in a loop. However, this case is rare since large thread-local arrays are known to impose high register pressure and are generally avoided (neither the Falcor nor the NVDIFFREC codebase use such arrays). We consider this an implementation limitation, rather than a fundamental one, that can be fixed if necessary by applying existing approaches (e.g., Dex [Paszke et al. 2021]).

8 CONCLUSION

We present SLANG.D, an extension that turns Slang into a fully differentiable shading language. Shading languages provide a natural imperative, per-thread programming model that fits well with the rendering logic. The integration of automatic differentiation (AD) into a shading language allows programmers to efficiently develop new *differentiable* renderers within their preferred programming model. Additionally, they can seamlessly transform an existing renderer targeting hardware graphics APIs into a differentiable one, by leveraging hundreds of thousands of lines of pre-existing shader and system code.

SLANG.D demonstrates that by treating AD as a first-class language feature, and by conducting a holistic co-design of language features and compiler backend, we can achieve a substantial advancement in expressiveness, performance and usability of an AD system. We believe that SLANG.D can serve as the bridge to connect machine learning and traditional rendering by lowering the effort required to integrate powerful rendering systems into a machine learning workflow. We hope that the method we have adopted to create SLANG.D can provide useful insights for incorporating AD to languages and tools in other domains.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 2105806, 2238839, CCF-1846502 and 2313024, and an NVIDIA Graduate Fellowship. We also thank Chris Wyman, Kayvon Fatahalian, Wesley Chang, Xuanda Yang, and the anonymous reviewers for their helpful feedback & proofreading.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. 2019. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation* 11, 1 (2019), 1–36.

Autodesk. 2023. Autodesk Aurora Path Tracing Renderer. https://github.com/Autodesk/Aurora

Sai Praveen Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased warped-area sampling for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 1–18.

Bradley Bell. 2003. CppAD: A Package for Differentiation of C++ Algorithms. http://www.coin-or.org/CppAD/

Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating a tensor language. arXiv preprint arXiv:2008.11256.

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

Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A type-based multi-stage programming framework for code generation in C++. In *Code Generation and Optimization.* 39–51.

Brent Burley. 2012. Physically-based shading at Disney. In *SIGGRAPH Course Notes. Practical physically-based shading in film and game production.*

Brent Burley. 2015. Extending the Disney BRDF to a BSDF with integrated subsurface scattering. In *SIGGRAPH Course Notes. Physically Based Shading in Theory and Practice.*

Petrik Clarberg, Simon Kallweit, Craig Kolb, Pawel Kozlowski, Yong He, Lifan Wu, Edward Liu, Benedikt Bitterli, and Matt Pharr. 2022. Real-Time Path Tracing and Beyond. HPG 2022 Keynote.

R. L. Cook and K. E. Torrance. 1982. A Reflectance Model for Computer Graphics. *ACM Trans. Graph.* 1, 1 (1982), 7–24.

Zachary Devito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. 2017. Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging. *ACM Trans. Graph.* 36, 5 (2017), 171:1–171:27.

Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *International Conference on Functional Programming* (2018).

Conal M Elliott. 2009. Beautiful differentiation. *SIGPLAN Not. (Proc. ICFP)* 44, 9 (2009), 191–202.

Theresa Foley. 2022. Managing Shader Code Complexity in NVIDIA Omniverse. https://www.linkedin.com/pulse/managing-shader-code-complexity-nvidia-omniverse-tim-foley/

T. Foley and Pat Hanrahan. 2011. Spark: modular, composable shaders for graphics hardware. *ACM Trans. Graph. (Proc. SIGGRAPH)* 30, 4 (2011), 1–12.

Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Trans. Math. Softw.* 22, 2 (jun 1996), 131–167.

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives.* Society for Industrial and Applied Mathematics.

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.* 39, 3 (2013), 20:1–20:43.

Jon Hasselgren, Nikolai Hofmann, and Jacob Munkberg. 2022. Shape, Light, and Material Decomposition from Images using Monte Carlo Rendering and Denoising. In *Advances in Neural Information Processing Systems.*

Jon Hasselgren, Jacob Munkberg, Jaakko Lehtinen, Miika Aittala, and Samuli Laine. 2021. Appearance-Driven Automatic 3D Model Simplification. In *Eurographics Symposium on Rendering.*

Yong He, Kayvon Fatahalian, and T. Foley. 2018. Slang: Language Mechanisms for Extensible Real-Time Shading Systems. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4, Article 141 (2018), 13 pages.

Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. 2022. Sparsity-specific code optimization using expression trees. *ACM Trans. Graph.* 41, 5 (2022), 1–19.

Robin J. Hogan. 2014. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.* 40, 4 (2014), 26:1–26:16.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *International Conference on Learning Representations* (2020).

Jan Hückelheim and Laurent Hascoët. 2022. Automatic differentiation of parallel loops with formal methods. In *International Conference on Parallel Processing.* 1–11.

Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. 2019. Automatic differentiation for adjoint stencil loops. In *International Conference on Parallel Processing.* 1–10.

Mathieu Huot and Amir Shaikhha. 2022. Denotationally Correct, Purely Functional, Efficient Reverse-mode Automatic Differentiation. arXiv preprint arXiv:2212.09801.

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures.* 319–338.

Homan Igehy. 1999. Tracing Ray Differentials. In *SIGGRAPH.* 179–186.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. DR. JIT: a just-in-time compiler for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 41, 4 (2022), 1–19.

Ulrik Jørring and William L Scherlis. 1986. Compilers and staging transformations. In *Principles of programming languages.* 86–96.

Simon Kallweit, Petrik Clarberg, Craig Kolb, Tom'aš Davidovič, Kai-Hwa Yao, Theresa Foley, Yong He, Lifan Wu, Lucy Chen, Tomas Akenine-Möller, Chris Wyman, Cyril Crassin, and Nir Benty. 2022. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations.*

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A Eisenberg, and Andrew Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–30.

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular primitives for high-performance differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 1–14.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization.* 75–86.

Tzu-Mao Li. 2018. *redner: Differentiable rendering without approximation.* https://github.com/BachiLi/redner

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018a. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.

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. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13.

Hsueh-Ti Derek Liu, Michael Tao, and Alec Jacobson. 2018. Paparazzi: Surface Editing by Way of Multi-view Image Processing. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 221:1–221:11.

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 228.

Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. 2021. Thallo–scheduling for high-performance large-scale non-linear least-squares solvers. *ACM Trans. Graph.* 40, 5 (2021), 1–14.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *European Conference on Computer Vision.*

William Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. *Advances in Neural Information Processing Systems* 33 (2020), 12472–12485.

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 *International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–16.

William S Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. 2022. Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In *International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–18.

Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. 2022. Extracting Triangular 3D Models, Materials, and Lighting From Images. In *Conference on Computer Vision and Pattern Recognition*. 8280–8290.

NVIDIA. 2023. NVIDIA RTX Remix. https://www.nvidia.com/en-us/geforce/rtx-remix/

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8024–8035.

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, Article 88 (2021), 29 pages.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1–7:36.

Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: generating renderers without writing a generator. *ACM Trans. Graph. (Proc. SIGGRAPH)* 38, 4 (2019), 1–12.

Alexey Radul, Adam Paszke, Roy Frostig, Matthew Johnson, and Dougal Maclaurin. 2022. You only linearize once: Tangents transpose to gradients. arXiv preprint arXiv:2204.10923.

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. (Proc. SIGGRAPH)* 31, 4 (jul 2012), 32:1–32:12.

Patrick Schmidt, Janis Born, David Bommes, Marcel Campen, and Leif Kobbelt. 2022. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. *Comput. Graph. Forum (Proc. SGP)* 41, 5 (2022), 113–124.

Kerry A Seitz Jr, T. Foley, Serban D Porumbescu, and John D Owens. 2019. Staged metaprogramming for shader system development. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 1–15.

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: Computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.* 5, POPL, Article 3 (2021), 31 pages.

Walid Taha. 2004. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*. 30–50.

Yusuke Takimoto, Hiroyuki Sato, Hikari Takehara, Keishiro Uragaki, Takehiro Tawara, Xiao Liang, Kentaro Oku, Wataru Kishimoto, and Bo Zheng. 2022. Dressi: A Hardware-Agnostic Differentiable Renderer with Reactive Shader Packing and Soft Rasterization. *Comput. Graph. Forum (Proc. Eurographics)* 41, 2 (2022), 13–27.

Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. 2008. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Trans. Math. Softw.* 34, 4 (2008), 18.

Delio Vicini, Sébastien Speierer, and Wenzel Jakob. 2021. Path Replay Backpropagation: Differentiating Light Paths using Constant Memory and Linear Time. *Transactions on Graphics (Proceedings of SIGGRAPH)* 40, 4 (2021), 108:1–108:14.

Yu. M. Volin and G. M. Ostrovskii. 1985. Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis. *Computers and Mathematics with Applications* 11 (1985), 1099–1114.

Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. 2019. The differentiable curry. In *Program Transformations for ML Workshop at NeurIPS*.

Richard Wei, Dan Zheng, Marc Rasi, and Bart Chrzaszcz. 2021. Differentiable Programming Manifesto. https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md

Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. 2022. Aδ: Autodiff for Discontinuous Programs - Applied to Shaders. *ACM Trans. Graph. (Proc. SIGGRAPH)* 41, 4, Article 135 (2022), 24 pages.

Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report. Microsoft Research.

Zihan Yu, Cheng Zhang, Derek Nowrouzezahrai, Zhao Dong, and Shuang Zhao. 2022. Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 41, 6 (2022), 1–16.

Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. 2021. Monte Carlo Estimators for Differential Light Transport. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4 (2021), 78:1–78:16.

Kai Zhang, Fujun Luan, Zhengqi Li, and Noah Snavely. 2022. IRON: Inverse Rendering by Optimizing Neural SDFs and Materials from Photometric Images. In *IEEE Conf. Comput. Vis. Pattern Recog.*

Shuang Zhao, Ioannis Gkioulekas, and Sai Bangaru. 2021. Physics-Based Differentiable Rendering. In *CVPR Tutorial*.

Shaokun Zheng, Zhiqian Zhou, Xin Chen, Difei Yan, Chuyan Zhang, Yuefeng Geng, Yan Gu, and Kun Xu. 2022. LuisaRender: A High-Performance Rendering Framework with Layered and Unified Interfaces on Stream Architectures. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 41, 6, Article 232 (2022), 19 pages.

A CODE STYLE COMPARISON: SIMT MODEL VS. NDARRAY MODEL

We use the snippet from the WAR algorithm in Fig. 12 to argue that the SIMT programming model is a better fit for differentiable rendering, compared the NDArray model employed by PyTorch and Dr.JIT. The NDArray model was originally proposed to express *interdependent* array operations (also referred to as "bulk-synchronous" operations) such as large matrix multiplications that cannot be concisely expressed in the SIMT style. However, this convenience comes at the expense of losing divergent branching for different elements, which must be now expressed through divergent data-flow with masks and `select` operators. Loops also need to be expressed with side-band constructs that explicitly declares loop state. As a result, authoring a differentiable renderer in the NDArray programming model is not as straightforward.

Renderers overwhelmingly use *independent* per-element operations with intricate control-flow and rarely need to synchronize their computation across the full image/data. We therefore argue, using Fig. 12 as an example, that the SIMT style is a better fit for writing differentiable renderers, allowing natural expression of control-flow without working with masks or side-band constructs. Further, the SIMT provides fine-grained synchronization features to exchange or reduce data at the thread-group level, which is more valuable for differentiable renderers than global synchronization over all threads. As an example: Sec. 6.2 takes advantage of SIMT features by using `WaveActiveSum` to speed up gradient aggregation without resorting to a separate kernel launch.

B SLANG.D EASE-OF-USE FEATURES

B.1 PyTorch Interoperation using slangpy

Many applications need to interact with deep learning systems. We created the slangpy python package that can create a PyTorch-compatible module from SLANG.D code with only a few lines:

```
import slangpy

shader = slangpy.loadModule(
   'mydiffshader.slang',   # Module name
   defines={'MY_PARAM': 2})  # Specialization parameters

output = shader.myfunc(myTorchTensor)
```

Where `myfunc` is a SLANG.D function defined in the `mydiffshader` file. This Python package uses the SLANG.D compiler to emit both CUDA and C++ binding code from a Slang module, and feeds the generated source into PyTorch's plugin system. To interop with PyTorch's tensor objects, SLANG.D provides a `TensorView` type that works directly with a PyTorch tensor's memory buffer, independent of the layout. This is in contrast to memory-managed AD systems like Dr.JIT that require the tensors to be formatted into a structure

```
// Every value is an array, including the index
idx = UInt32(0)
// Must explicitly define loop condition and state
// in a side-band structure.
loop = Loop("myloop", idx < 8,
          lambda: idx, wt, wtDir)
while (loop):
   auxRay = sampleAuxRay(ray);

   auxRayF = reflect(ray, auxRay);
   auxRay = select(idx % 2 == 0, auxRay, auxRayF)

   isect = traceRay(auxRay)
   // 'if' conditions emulated using masks
   mask = dr.and(isect.isValid(), mask)

   // Compute wt_i, wtDir_i ...
   wt = wt + select(mask, wt_i, 0)
   wtDir = wtDir + select(mask, wtDir_i, 0)
   idx = idx + 1
//...
```

Embedded NDArray Programming Model (PyTorch, Dr. JIT)

```
// Loop state is obtained by data-flow analysis,
// therefore no additional annotations required.

for (uint idx = 0; idx < 8; idx++) {
   // Values are per-thread, not nd-array
   Ray auxRay = sampleAuxRay(ray);

   // Since all values are per-thread,
   // we can use if conditions and breaks
   if (idx % 2 == 0)
      auxRay = reflect(ray, auxRay);

   Intersection isect = traceRay(auxRay);

   if (!isect.isValid()) break;

   // Compute wt_i, wtDir_i ...
   wt += wt_i;
   wtDir += wtDir_i;
}
//...
```

Shader SIMT Programming Model (SLANG.D)

Fig. 12. Coding Style Comparison We compare code style and quality by showing an example snippet from the WAR algorithm implemented in the NDArray programming style (left) and in the SIMT style (right). We argue that the NDArray model was originally proposed to express *inter-dependent* array operations such as large matrix multiplications that cannot be concisely expressed in the SIMT style. This comes at the expense of per-element control flow, which must be expressed using masks. However, differentiable renderers overwhelmingly use *independent* computation, containing much more intricate control-flow and rarely ever need to synchronize across threads. We show through this snippet that the SIMT style is a better fit for writing differentiable renderers, avoiding masks, explicit loop state, and more.

of arrays (SoA) format. Differentiable renderers written in SLANG.D can use its custom derivative primitives (Sec. 4.3) to load/store data in any layout and avoid the overhead of data marshalling.

slangpy enabled us to create both micro-benchmarks in Sec. 6.2 within an hour. We also used slangpy to port the kernels in nvdiffmodeling and nvdiffrec discussed in Sec. 6.1.3.

B.2 Debugging Differentiable Shaders with Custom Derivatives

Many GPU-based languages offer some way to debug code through some per-thread version of C's `printf`. Slang also offers a `print()` function on targets that support it (e.g., HLSL, CUDA). Being able to debug code is especially important when writing large scale renderers with many components, and the same is true for differentiable ones. We used SLANG.D's support for *arbitrary* custom derivatives to our advantage by using them to print derivatives.

Here is a snippet demonstrating derivative print functions:

```
[ForwardDerivative(fwd_myPrint)]
void myPrint(String msg, float val) {
   print(msg, val);
}

void fwd_myPrint(
   String msg,
   DifferentialPair<float> val) {
   print(msg, val.d);
}
```

Calling `myPrint("%f", x)` will print the value of `x` in the primal function and the value of `x.d` in the derivative propagation function.

The derivative printing method `fwd_myPrint` can be provided another custom gradient to debug higher-order derivatives of `val` which was pivotal to our debugging process for the reparameterized differentiable renderer in Sec. 6.1.2.

B.3 Inverted Custom Derivative Attributes for Integration with Existing Codebases

Integrating a differentiable renderer into a traditional codebase requires sharing common code that needs user-defined derivatives. However our `[ForwardDerivative]` and `[BackwardDerivative]` attributes are annotated on the primal functions (referencing the derivatives). This pattern of annotation would require modifying large portions of the code shared between the differentiable and traditional parts of the codebase.

Instead, we found it cleaner to provide a second set of attributes: `[ForwardDerivativeOf]` and `[BackwardDerivativeOf]`, which establish the same relationship — only by annotating the derivative, rather than primal function.

We understand that this could be seen as an anti-pattern. (via non-obvious overrides that can come from a different file entirely) In practice, the benefits of the separation of concerns outweighed the downsides, which we mitigated by augmenting Slang's Intellisense extension to highlight such overridden derivatives.

C ADDITIONAL BACKGROUND

C.1 Automatic Differentiation

We review some common terminology in automatic differentiation and refer the readers to Griewank and Walther [2008] for a comprehensive treatment of automatic differentiation techniques.

Automatic differentiation applies the chain rule to propagate derivatives in programs. Crucially, it propagates derivatives while creating necessary intermediate variables, so that the computation time remains efficient. The two popular differentiation methods, *forward-mode* and *reverse-mode* differentiation correspond to a function's derivative and its adjoint. Computationally, they differ in the way they traverse the program and cache intermediate variables.

Forward-mode differentiation. Given a function $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, forward-mode differentiation produces a *total derivative* function $\mathrm{d}\mathbf{y} = Df(\mathbf{x}, \mathrm{d}\mathbf{x}) : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m$, such that $\mathbf{y} + \mathrm{d}\mathbf{y}$ is the closest linear approximation to $f(\mathbf{x} + \mathrm{d}\mathbf{x})$. Forward mode associates each scalar variable with a scalar differential, and propagates the differential $\mathrm{d}\mathbf{x}$ from inputs to outputs using standard differentiation rules. The time and space complexity of the total derivative Df are the same as the primal function f.

Reverse-mode differentiation. Differential operators like the gradient are more naturally expressed as the transpose (aka. adjoint) of the total derivative. ($\nabla_{\mathbf{x}} f = D^T f(\mathbf{x}, 1)$) Reverse mode instead computes a function $\mathrm{d}\mathbf{x} = D^T f(\mathbf{x}, \mathrm{d}\mathbf{y}) : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$, where this *adjoint derivative* $D^T f$ now takes the differentials of the *output* $\mathrm{d}\mathbf{y}$ and emits the differentials of the input $\mathrm{d}\mathbf{x}$. Computationally, this requires running the program first *forwards* and then *backwards* (the transpose), reversing all control flow in the process. Reverse mode preserves the time complexity of the primal function, but requires as much *space* as the original function took *time*—which may be a very large increase in space usage.

Checkpointing. A commonly used remedy for the reverse-mode space blowup is *checkpointing* [Volin and Ostrovskii 1985]: instead of remembering all intermediate values, we can recompute some of the forward computation results on-the-fly as we run the function backwards. Checkpointing trades between memory and time by deciding how much of the forward computation should be memoized.

C.2 Design Space of Automatic Differentiation Systems

We mainly focus on systems that support massive parallelism and omit the more traditional serial systems (e.g., [Andersson et al. 2019; Bell 2003; Griewank et al. 1996; Hascoet and Pascual 2013; Hogan 2014; Pearlmutter and Siskind 2008; Utke et al. 2008]) and the recent theoretical endeavors of the semantics (e.g., [Elliott 2018, 2009; Krawiec et al. 2022; Sherman et al. 2021]).

Automatic differentiation systems can be broadly seen as compilers/interpreters that transform/interpret code into its derivatives. Specifically, they act as *domain-specific languages* (DSLs) that specialize at computing derivatives. Imagine putatively adding automatic differentiation to an existing general-purpose language. Doing so requires visible extensions to allow user direction of which code to differentiate. Furthermore, automatic differentiation subtly changes the semantics of a language, necessitating decisions about how differentiation interacts with existing langauge features.

Front-end strategy. One way to classify DSLs is loosely, based on how they design their frontends — i.e., the *syntax* rather than *execution model.* A DSL can be stand-alone with its own syntax, or it can be embedded in a host language. *Shallow-embeddings* avoid materializing an intermediate representation (IR). Many forward-mode automatic differentiation systems are implemented this way, by eagerly computing differentials along with the primal values (e.g., via operator overloading). By contrast, *Deep-embeddings* use host-language constructs to construct and materialize an IR of the DSL program. For example, PyTorch [Paszke et al. 2019] embeds itself in Python as classes and functions, and obtains an IR – the computational graph – by tracing through these classes/functions.

Most automatic differentiation systems are either stand-alone DSLs (e.g., Enzyme [Moses and Churavy 2020; Moses et al. 2021, 2022], Dex [Paszke et al. 2021]) or deeply-embedded in a host language (e.g., PyTorch, JAX [Bradbury et al. 2018], Dr. JIT [Jakob et al. 2022], DiffTaichi [Hu et al. 2020]). Shallow embedding is easy to implement, but implementers quickly find out that they will need a non-trivial data structure to manipulate derivative computation. For example, many classical operator-overloading based automatic differentiation systems (e.g., CppAD [Bell 2003] or ADOL-C [Griewank et al. 1996]) maintain a tape of instructions as their IR.

A challenge for deep-embedding DSL is to capture complex program structures, such as loops and branches [Brahmakshatriya and Amarasinghe 2021]. These systems often have to invent new syntax for describing loops and branches, as the control flows in the host language has different semantics than the control flows in the DSL.

Our system belongs to the class of stand-alone DSLs. Even though we build on an existing language (Slang), we modify Slang's compiler to compile our new language. This allows us to differentiate shader code with minimal changes. The challenge is that our system now needs to interact with all language features, including control flows, generics, and polymorphism, in Slang.

Notably, Dressi-AD [Takimoto et al. 2022], while building their system on top of GLSL, focuses on deferred shading with rasterization and does not describe the handling of language features such as the ones mentioned above.

Staged execution. A common strategy for compiling embedded DSLs is to make use of *staged execution*, aka. *partial evaluation*, aka. *specialization* [Jørring and Scherlis 1986; Taha 2004]. Given a function `f(x, y)` and a specific value `x0`, partial evaluation produces a (potentially optimized) function `g(y)` such that `g(y)=f(x0, y)`. Compilers can be described as specialization of an interpreter `interp(prog, input)` to a particular program `prog0` – a two-stage execution. However, embedded DSLs often make use of non-standard or additional stages.

For example, many automatic differentiation sytems rely on *tracing*, where (stage 1) a host-program is run to produce a DSL IR, and only later executed (stage 2). Describing the behavior of a program written in such a system requires clarifying whether or not a variable is specialized (i.e., evaluated) during tracing (a stage 1 variable) or as part of the traced program (a stage 2 variable). Similarly, if the DSL/tracing system seeks to capture control flow (loops, if-statements, dynamic dispatch), then new (stage 2) versions of host (stage 1) language constructs have to be added. As a result, users of tracing-based automatic differentiation systems must learn to perform a limited form of meta-programming when they reason about such distinctions. We avoid this approach in part because it is more complicated for programmers to reason about.

When the tracing of a host metaprogram executes through loops and branches in the host language it produces a specialized, unrolled compute graph, i.e., the DSL program. This DSL program then must be evaluated. While the "forward evaluation" of the DSL program and the host program are often interleaved (as in the case of default PyTorch and Dr. JIT), they need not be. For example, the AOT autograd[11] module in PyTorch allows one to generate the DSL program "ahead-of-time" and execute it separtely from the host program.

Thus, even in the context of tracing, static compilation vs. dynamic interpretation is just a question of staging. It is even possible to support multiple stages. For example, CUDA programs are often "compiled" statically to PTX virtual assembly, and then "JIT compiled" to the binary code of the target architecture by the CUDA driver. Finally, they are executed.

Specialization for the purposes of optimization is hence not exclusive to tracing. Staging [He et al. 2018; Pérard-Gayot et al. 2019; Seitz Jr et al. 2019; Zheng et al. 2022] allows us to specialize a shader program to a specific type or values even before any tracing happens, by statically compiling an abstract program supplied with a concrete type or input. Tracing is also not limited to low-cost compiler optimization, as the execution of the partial evaluated function can be separated from the host code.

In our setting, since we directly build on the Slang language and compiler, we opt to not doing tracing. However, thanks to Slang's generics language feature, our system still allows specialization to a known type or value. For example, we can specialize a "Uber BRDF" and its derivatives into one of its component, say, a diffuse BRDF.

Intermediate representation. The central decision of a language is how it chooses to represent programs. For example, one of the simplest representations of a program is a sequential *tape* that records the instructions and their arguments (e.g., `sin(x)` or `a + b`).

One major axis of variation in automatic differentiation IRs is whether or not control flow is captured and represented. One option is to simply capture loops and branches (if-statements) directly. Another is to omit control flow from the IR altogether (e.g., when the IR is merely a tape). Many tracing systems that started by omitting control flow from IRs often have to retroactively add it back in (e.g., `tf.while_loop` and `dr.cuda.Loop`). The client programmers must now learn to distinguish between host-language vs. DSL-level control flow constructs (one of which executes during the metaprogramming stage and the other during the DSL execution stage).

Another major axis of variation is how side-effects are handled. In particular, there has been much concern in recent years over trying to find functional IRs, that are completely without side effects or with well-controlled ones, to support efficient AD [Bernstein et al. 2020; Huot and Shaikhha 2022; Paszke et al. 2021]. However, most systems rely on mutative updates, especially for reverse mode accumulations (notably, Pearlmutter and Siskind [2008]'s system relies on a += operation to update the adjoints).

[11] https://pytorch.org/functorch/stable/aot_autograd.html

The IRs may be higher-level (mirroring the source code) or lower-level inside the compiler. Enzyme is notable for advocating the latter design for extending LLVM [Lattner and Adve 2004] with automatic differentiation features.

Automatic differentiation IRs are often quite primitive, and lack more complex typing or object-oriented language features (with notable exception of the Swift automatic differentiation work [Vytiniotis et al. 2019; Wei et al. 2021] and some work on higher-order function differentiation [Huot et al. 2020; Krawiec et al. 2022]). Our work incorporates the type system and object-oriented features in Slang into the IR directly, taking a similar approach to the ongoing Swift work, but applying it to shading languages.

Finally, another choice is whether the IR is closed under differentiation. If the differentiation produces code that can be not differentiable, it can cause troubles for higher-order or nested differentiation. Many automatic differentiation systems do not support higher-order differentiation because the code generated by differentiation is not the same IR before differentiation [Hu et al. 2020; Jakob et al. 2022]. Our system ensures closure of IR under differentiation and supports higher-order differentiation.

Program optimization. DSLs for automatic differentiation are distinguished by how they choose to optimize and compile their code, which largely follows from a characterization of the expected programs. The Deep Learning system design is one approach [Abadi et al. 2015; Paszke et al. 2019; Yu et al. 2014]. Deep learning pipeline runtimes tend to be dominated by a selected set of layers (fully connected, convolution, attention, etc.), whose implementations are supplied via highly-efficient hand-tuned kernels rather than being generated by the compiler. As a result, compilation is primarily focused on highly-localized peephole optimizations to fuse nearby computations into these key layers. For example, XLA (which serves as the compilation layer for JAX and TensorFlow) contains templated versions of key computation kernels to support variations through fusion. A variety of systems offer other approaches. For example, the automatic differentiation system of Halide [Li et al. 2018b; Ragan-Kelley et al. 2012] offers code transform for long-range fusion and a scatter-gather transform during the differentiation to avoid race condition. Opt [Devito et al. 2017] and Thallo [Mara et al. 2021], on the other hand, specialize for sparse-Hessian-vector-products where the sparsity is determined by the program structure. Our situation is characterized by the situation of shading languages: we avoid complex program analysis for global optimization, while providing sufficient flexibility to the users to achieve high-performance.

C.3 Shading Languages

In real-time rendering, shading languages were originally introduced to define the kernel code (shader) executed during the programmable stages of the hardware-accelerated graphics pipelines. Popular shading languages like HLSL, GLSL and the Metal Shading Language provide a Single-Instruction-Multiple-Threads programming model that maps natively to modern GPU architectures, which are optimized for the data-parallel nature of rendering workloads.

To support the extreme demand for high performance from real-time rendering applications, shading languages are designed to ensure that performance critical optimizations can be performed without relying on complex compiler analysis. For example, most shading languages do not provide access to pointers, and global memory read and write operations can only be done through explicit resource handles. Memory exposed by different resource handles are generally considered to be non-overlapping, so the compiler can perform aggressive optimizations without worrying about address aliasing. Operations on a read-only resource handle can also be safely considered as side-effect-free. Meanwhile, traditional language features that cannot be implemented efficiently on GPUs (such as recursive function calls, heap allocation, and dynamic lifetime management) are generally prohibited in shading languages.

The demand for high performance also drives applications to aggressively specialize shaders into many variants where each variant caters a specific use case. For example, when rendering a scene with three point lights, the rendering framework will ensure that a minimal amount of code is sent to GPU for execution. This is accomplished by creating a specialized shader that contains only the code to evaluate point lights and not other types of lights. Such specialization improves the runtime performance by enabling more code to be evaluated at compile time, and by removing unused code branches from the final shader kernel to reduce the maximum register consumption.

D DIFFERENTIATION OF TEXTURE SAMPLING

As a common operation in rendering, texture sampling is accelerated by the hardware and therefore exposed as an intrinsic operation in shading languages. Since texture sampling involves multiple global memory reads, `SLANG.D` does not automatically differentiate through these operations, as discussed in Sec. 4.3.

However, we can use `SLANG.D`'s *primal substitute* mechanism to provide a software implementation of texture sampling for automatic differentiation, and then use custom derivative functions to handle the gradient accumulation for individual texel loads.

To make it easy for existing shader code written against the built-in `Texture2D` type to propagate derivatives back to the textures, we can define a differentiable texture type completely in user code:

```
SLANG.D
struct MyDifferentiableTexture {
   Texture2D hwTextureHandle;
   Buffer dBuffer;

   [BackwardDerivative(WriteTexelDerivative)]
   float4 LoadTexel(uint2 location, uint lod) {
      hwTextureHandle.Load(location);
   }

   void WriteTexelDerivative(
      uint2 location, uint lod, float4 derivative) {
      // Atomic accumulate to dBuffer
   }

   [PrimalSubstitute(sample_ref)]
   float4 Sample(
      SamplerState s, float2 uv, float2 ddx, float2 ddy) {
      hwTextureHandle.Sample(s, uv, ddx, ddy);
   }

   [Differentiable]
   float4 sample_ref(
      SamplerState s, float2 uv, float2 ddx, float2 ddy) {
      // Software implementation of trilinear sampling ...
      // ... calls LoadTexel to read texels ...
   }
};
```

Developers can use `MyDifferentiableTexture` as a drop-in replacement of the built-in `Texture2D` type, and call the `Sample` method as usual. When the `SLANG.D` compiler's differentiation pass sees a call to `MyDifferentiableTexture.Sample`, it will instead differentiate through `sample_ref` as if the user code were calling `sample_ref`, thus propagating the derivative through our software trilinear sampling method all the way back to the accumulation buffer. We can then use a follow-up pass to process `dBuffer` and turn it into a texture containing the propagated derivatives.

To provide more details, we include the full shader code for differentiable trilinear sampling in the supplementary material as the `texture.slang` file.

E INFERENCE RULES OF FORWARD AND BACKWARD DERIVATIVE FUNCTION SIGNATURES

The general rule for determining the signature of a forward derivative function is to transform each differentiable parameter into a `DifferentialPair` that holds both the original parameter value and the derivative associated with the parameter for forward propagation. More specifically, the signature of its forward derivative function is determined using the following rules:

(1) If the return type `R` is differentiable, the forward derivative function will return `DifferentialPair<R>` that consists of both the computed original result value as well as the (partial) derivative of the result value. Otherwise, the return type is kept unmodified as `R`.
(2) If a parameter has type `T` that is differentiable, it will be translated into a `DifferentialPair<T>` parameter in the derivative function, where the differential component of the `DifferentialPair` holds the initial derivatives of each parameter with regard to their upstream parameters.

(3) All parameter directions are unchanged. For example, an `out` parameter in the original function will remain an `out` parameter in the derivative function.

The general rule for determining the signature of a backward derivative function is that a differentiable output o becomes an input parameter holding the partial derivative of a downstream output with regard to the this differentiable output, i.e. $\partial y/\partial o$); an input differentiable parameter i in the original function will become an output in the backward propagation function, holding the propagated partial derivative $\partial y/\partial i$ and any non-differentiable outputs are dropped from the backward propagation function. This means that the backward derivative function never returns any values computed in the original function.

More specifically, the signature of a backward derivative function is determined using the following rules:

(1) A backward derivative function always returns `void`.
(2) A differentiable `in` parameter of type `T` will become an `inout DifferentialPair<T>` parameter, where the original value part of the differential pair contains the original value of the parameter to pass into the back-prop function. The original value will not be overwritten by the backward derivative function. The propagated derivative will be written to the derivative part of the differential pair after the backward propagation function returns. The initial derivative value of the pair is ignored as input.
(3) A differentiable `out` parameter of type `T` will become an `in T.Differential` parameter, carrying the partial derivative of some downstream term with regard to the return value.
(4) A differentiable `inout` parameter of type `T` will become an `inout DifferentialPair<T>` parameter, where the original value of the argument, along with the downstream partial derivative with regard to the argument is passed as input to the backward derivative function as the original and derivative part of the pair. The propagated derivative with regard to this input parameter will be written back and replace the derivative part of the pair. The primal value part of the parameter will *not* be updated.
(5) A differentiable return value of type `R` will become an additional `in R.Differential` parameter at the end of the backward derivative function parameter list, carrying the result derivative of a downstream term with regard to the return value of the original function.
(6) A non-differentiable return value will be dropped from the derivative function signature.
(7) A non-differentiable `in` parameter will remain unchanged in the backward propagation function.
(8) A non-differentiable `inout` parameter will become an ‘in‘ parameter of the same type.

For example consider the following original function:

```
struct T : IDifferentiable {...}
struct R : IDifferentiable {...}
struct ND {} // Non differentiable

[Differentiable]
R original(
   T p0,
   out T p1,
   inout T p2,
   ND p3,
   out ND p4,
   inout ND p5);
```

The signature of its backward derivative function is:

```
void back_prop(
   inout DifferentialPair<T> p0,
   T.Differential p1,
   inout DifferentialPair<T> p2,
   ND p3,
   ND p5,
   R.Differential dResult);
```

Note that although `p2` is still `inout` in the backward propagation function, the backward derivative function will only write propagated derivative to `p2.d` and will not modify the primal value in `p2.p`.

F AN EXAMPLE OF EXPONENTIAL CODE EXPANSION WITHOUT CONSTRAINING `DIFFERENTIAL` TYPE

In Sec. 4.2, we stated that it is important to make the `Differential` associated type in `IDifferentiable` interface (Listing 3) to conform to this additional constraint:

`Differential.Differential=Differential`

Here, we provide an example showing the compiler will need to generate exponentially large amount of code without assuming this constraint. Imagine the user has defined three types `A1`, `A2`, `A3`, where

```
A1.Differential = A2
A2.Differential = A3
A3.Differential = A1
```

We notate that in a shorter form as `A1->A2->A3`. Similarly, assume there is another set of types `B1->B2`, and the compiler is asked to synthesize the `Differential` type for a product type `(A1, B1)`. In Slang, this can be represented as a struct type:

```
struct MyType : IDifferentiable {
   A1 field1;
   B1 field2;
}
```

By performing the synthesis structurally on each field, the compiler will generate a type `(A2, B2)` as the first-order `Differential`. But since the differential type itself must also be differentiable, the compiler must continue synthesizing the second-order differential type `(A3, B1)`. This process will continue until the newly synthesized differential type is equivalent to an existing one. In this particular case, the compiler will synthesize 6 types in total: `(A2, B2)`, `(A3, B1)`, `(A1, B2)`, `(A2, B1)`, `(A3, B2)`, `(A1, B1)`. As can be seen in this simple example, the compilation became an exponential process with respect to the number of fields in a product type.

G CONTROL-FLOW NORMALIZATION PASS

As discussed in Section 5.1, the purpose of the control-flow normalization step is to transform the CFG of a differentiable function into a reversible form, where jumps in the CFG are either to the next block in a sequential region, to different branches in an `if` or `switch` region, to the merge point at the end of each branch, or to the loop header at the end of a loop. In other words, if a CFG in reversible form is represented with structured control-flow primitives, there will be no `goto`, `continue` or `break` statements. There will also be only one `return` statement at the end of the function.

Control-flow normalization is done in four steps: `return` removal, `continue` removal, `break` removal and loop canonicalization. The first two steps removes any early `return`s and `continue` statements by transforming them into a `break`. The third step removes `break` statements by introducing a boolean flag tracking whether a `break` took place and guard the operations after the `break` statement with the boolean flag. Finally, we transform all loops into a single canonical form to simplify the implementation of the AD pass. We will use concrete examples to illustrate each transform.

G.1 `return` Removal

In this step, we rewrite functions with early `return`s into `break` statements. For example, consider the following code:

```
if (x < 1)
   return 0;
int y = x + 1;
return y;
```

We can rewrite this code by wrapping the entire function into a one-iteration loop, and replace all `return`s into `break`s out of the one-iteration loop:

```
int returnVal;
while (1) {
   if (x < 1) {
      returnVal = 0;
      break;
   }
   int y = x + 1;
   returnVal = y;
   break;
}
return returnVal;
```

G.2 `continue` Removal

In this step, we remove all the `continue` jumps from the CFG, using the same idea of using `break`s out of one-iteration loops to replace `continue`s. For example, consider the following code:

```
int sum = 0;
for (int i = 0; i < n; i++) {
   if (i % 2 == 0)
      continue;
   sum += i;
}
```

We can wrap the loop body with a new one-iteration loop, so the `continue` can be rewritten into a `break`:

```
int sum = 0;
for (int i = 0; i < n; i++) {
   while (1)  {
      if (i % 2 == 0)
         break;
      sum += i;
      break;
   }
}
```

Note that if the original loop contains `break` statements, the original `break` will become a multi-level break that jumps to the end of the outer loop. This is allowed in the Slang IR.

G.3 `break` Removal

After the first two steps, the only jumps that are not inherent to control-flow constructs (e.g. `for`, `if` or `switch`) are the jumps representing a `break` statement. To remove these `break`s, we insert a boolean flag tracking whether the execution of the current sequential region has been terminated by a `break`, so that the `break` can be rewritten to set the termination flag to `true`. The operations following the `break` will be guarded by the termination flag. For example, the `while` loop in the previous example will become:

```
...
bool terminate_flag = false;
while (1) {
   if (i % 2 == 0)
      terminate_flag = true;
   if (!terminate_flag) {
      sum += i;
      terminate_flag = true;
   }
}
```

After these normalization steps, functions with early `return`s, `break` and `continue` statements will be transformed into a simpler form without these constructs, so we can easily reverse the control flow by simply arranging the control-flow regions in the reverse order.

G.4 Loop Canonicalization

To allow the automatic differentiation passes to work on all types of loops regardless of whether they are originally defined by a `for`, `while` or `do-while` statement in user code, the CFG normalization pass canonicalizes all loops into the form of:

```
loopHeaderBlock:
    condition = condition of whether loop should continue
    cbranch condition, loopBodyBlock, loopExitBlock
loopBodyBlock:
   Loop body logic
   branch continueBlock
continueBlock:
   branch loopHeaderBlock
loopExitBlock:
   End of loop
```

Note that in this canonical form, the only back-jump inside a loop is defined in `continueBlock`, and the only jump out of the loop is the `cbranch` instruction in `loopHeaderBlock`. After SSA transformation, all loop state variables (variables that are updated during a loop iteration) will become ϕ instructions in `loopHeaderBlock`.

At the end of CFG normalization pass, all normalized functions will have only one `return` point, and all loops will have only one exit point.

H DIFFERENTIABLE WARP FUNCTION IMPLEMENTATION

As discussed in Sec. 6.1.2, SLANG.D enables us to implement warped-area reparameterization by building the warp function, using a nested forward-mode pass instead of hand-coding. Listing 11 shows a differentiable version of $\mathcal{V}^{(\text{harmonic})}$ that is computed by taking the weighted mean of points *attached* to the geometry. Listing. 12 shows how the `reparameterize()` method uses `fwd_diff` to compute the reparameterization.

```
                                                 SLANG.D
[Differentiable]
float2 warpedSample(float2 uv) {
   //...
   Ray ray = sampleToRay(uv);
   for (i = 0; i < auxCount; i++) {
      Ray auxRay = sampleAuxRay(Ray(dir, o));
      float3 pt = intersect(auxRay);

      float2 attachedUV = projectToScreen(pt);
      float wt = harmonicWt(auxRay, pt);

      totalAttachedUV += wt * attachedUV;
      totalWt += wt;
   }
   float2 meanUV = totalAttachedUV / totalWt;
   return meanUV - detach(meanUV);
}
```

Listing 11. Primal logic to reparameterize a sample according to the warp $\mathcal{V}^{(\text{harmonic})}$ described by Bangaru et al.[2020]. As described by their appendix B, the divergence $\nabla.\mathcal{V}$ is equivalent to the Jacobian of this function, and can be computed by placing multiple calls to `fwd_diff(warpedSample)`. Note that `detach()` is similar to PyTorch's `stop_gradient()`: it serves as the identity function in the absence of differentiation, but transforms to a 0 differential when differentiated, effectively stopping the propagation of gradients.

```
                                                 SLANG.D
[Differentiable]
float2x2 infinitesimal(float2x2 xy)
{ return xy - detach(xy); }

// Note that inspite of using AD, this function
// can further be used in a differentiable pipeline
[Differentiable]
float2 reparameterize(float2 uv, out float wt) {
   // Map uv to reparamterized sample
   float2 w_uv = warpedSample(uv);

   // Compute Jacobian of the mapping
   float2 w_uv_dx = fwd_diff(warpedSample)(
      diffPair(w_uv, float2(1.0, 0.0))).d;
   float2 w_uv_dy = fwd_diff(warpedSample)(
      diffPair(w_uv, float2(0.0, 1.0))).d;

   float2x2 J = identity<2>() + infinitesimal(
        float2x2(w_uv_dx.x, w_uv_dy.x,
              w_uv_dx.y, w_uv_dy.y));

   wt = determinant(J);
   return uv + w_uv;
}
```

Listing 12. Primal logic to reparameterize a screen space sample. `reparameterize()` uses SLANG.D's forward-mode AD to elegantly construct the Jacobian of the warp mapping and computes the reparameterization weight. Note that this requires d invocations for a d-dimensional mapping. Our practical implementation uses an array of d replica samples to avoid tracing auxiliary rays multiple times.