LuisaRender: A High-Performance Rendering Framework with Layered and Unified Interfaces on Stream Architectures

SHAOKUN ZHENG, BNRist, Department of CS&T, Tsinghua University, China
ZHIQIAN ZHOU, BNRist, Department of CS&T, Tsinghua University, China
XIN CHEN, BNRist, Department of CS&T, Tsinghua University, China
DIFEI YAN, BNRist, Department of CS&T, Tsinghua University, China
CHUYAN ZHANG, BNRist, Department of CS&T, Tsinghua University, China
YUEFENG GENG, Recreate Games, China
YAN GU, University of California, Riverside, United States
KUN XU*, BNRist, Department of CS&T, Tsinghua University, China

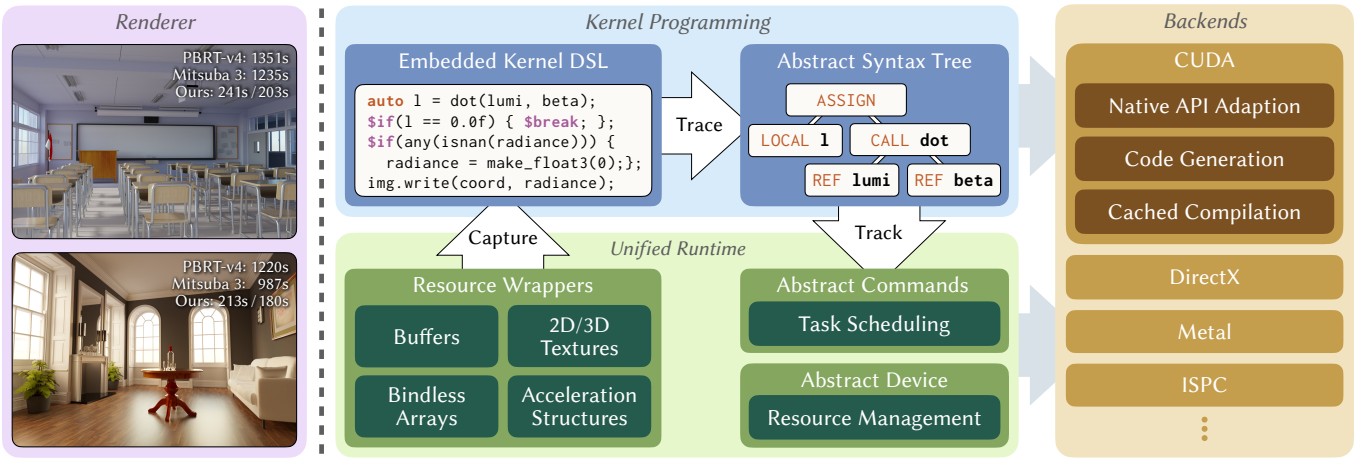


Fig. 1. The system architecture of LuisaRender (right) and a rendering application built atop it (left). Our layered framework provides a kernel programming language and a unified runtime on various backends. We compare the performance of our renderer with PBRT-v4 and Mitsuba 3 on two scenes, *Classroom* and *Living Room*. Results are shown on the left, with the rendering time of PBRT-v4 (CUDA, wavefront), Mitsuba 3 (CUDA, mega-kernel), and our renderer (CUDA and DirectX, wavefront). Both scenes are spectrally rendered on RTX-3080Ti with 16 bounces and 16384spp. Our renderer is about 5.5× faster than PBRT-v4 and 4.5× faster than Mitsuba 3 on the same CUDA backend, and is even faster on DirectX.

The advancements in hardware have drawn more attention than ever to high-quality offline rendering with modern stream processors, both in the industry and in research fields. However, the graphics APIs are fragmented and existing shading languages lack high-level constructs such as polymorphism, which adds complexity to developing and maintaining cross-platform high-performance renderers. We present LuisaRender[1], a high-performance rendering framework for modern stream-architecture hardware. Our main contribution is an expressive C++-embedded DSL for kernel programming with JIT code generation and compilation. We also implement a unified runtime layer with resource wrappers and an optimized Monte Carlo renderer. Experiments on test scenes show that LuisaRender achieves much higher performance than existing research renderers on modern graphics hardware, e.g., 5–11× faster than PBRT-v4 and 4–16× faster than Mitsuba 3.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Parallel programming languages**; • **Software and its engineering** → **Domain specific languages**.

Additional Key Words and Phrases: rendering framework, stream architecture, cross-platform renderer

*Kun Xu is the corresponding author.

Authors' addresses: Shaokun Zheng, BNRist, Department of CS&T, Tsinghua University, Beijing, China, zsk20@mails.tsinghua.edu.cn; Zhiqian Zhou, BNRist, Department of CS&T, Tsinghua University, Beijing, China, zhouzq18@mails.tsinghua.edu.cn; Xin Chen, BNRist, Department of CS&T, Tsinghua University, Beijing, China, chenxin18@mails.tsinghua.edu.cn; Difei Yan, BNRist, Department of CS&T, Tsinghua University, Beijing, China, ydf18@mails.tsinghua.edu.cn; Chuyan Zhang, BNRist, Department of CS&T, Tsinghua University, Beijing, China, chuyan-z19@mails.tsinghua.edu.cn; Yuefeng Geng, Recreate Games, Shanghai, China, maxwellgeng@outlook.com; Yan Gu, University of California, Riverside, Riverside, United States, ygu@cs.ucr.edu; Kun Xu, BNRist, Department of CS&T, Tsinghua University, Beijing, China, xukun@tsinghua.edu.cn.

0730-0301/2022/12-ART232
https://doi.org/10.1145/3550454.3555463

[1] LuisaRender is open-source with applications at https://github.com/LuisaGroup.

ACM Reference Format:
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.* 41, 6, Article 232 (December 2022), 19 pages. https://doi.org/10.1145/3550454.3555463

1 INTRODUCTION

The demand to quickly render increasingly complex scenes is drawing attention to the development of high-performance rendering systems. Among the techniques, stream-processing architectures, typically GPUs with massive parallelism, are a prospective choice. Graphics hardware and software frameworks are emerging and evolving, yet accompanied by new challenges.

A major challenge in exploiting the stream-processing architectures is the current design of graphics APIs and shading languages. Modern graphics APIs tend towards explicitness and controllability and the companion shading languages lack enough expressiveness and inter-operability with the host language [He et al. 2018, 2017]. For example, users are required to one-by-one bind arguments to the compute shader (sometimes necessarily translating the data layouts, too), and then manually inform the driver to be prepare and map the resources. Also, the fragmentation of APIs and shading languages complicates the design of cross-platform applications, making the development and mantainance painful and error-prone.

Various domain-specific frameworks and languages are proposed to ease the programming on modern stream-processing hardware. For example, Taichi [Hu et al. 2019] is an easy-to-use and efficient *domain-specific language* (DSL), mainly designed for physical simulation. Halide [Ragan-Kelley et al. 2012] and follow-up work aimed at automatic scheduling for image processing tasks, which adopts a functional programming model at the level of image buffers. Though not primarily targeted at rendering, their insightful designs, such as embedded DSLs and execution scheduling, inspired us in this work.

Targeting rendering tasks, Mitsuba 2 [Nimier-David et al. 2019] is a retargetable system on Enoki [Jakob 2019]. It uses an array-based programming model, which achieves good performance on CPUs with small vectorization widths but results in much overhead on the GPU side—mainly due to the memory traffic of arrays and the thread divergence from the manually implemented array masks. The follow-up work, Dr.JIT [Jakob et al. 2022a], alleviates the memory bandwidth pressure by fusing computation graph nodes into mega-kernels. Mitsuba 3 [Jakob et al. 2022b] is the successor renderer atop Dr.JIT and also has a better performance. PBRT [Pharr et al. 2016] is a widely appreciated educational renderer and recently has introduced GPU support into its latest version. The implementation is based on the straightforward source-level code sharing between CUDA and C++, which well serves its principles of clarity and simplicity but sets a limit on the portability to other platforms.

We propose LuisaRender, a rendering framework that seeks to balance the seemingly ever-conflicting pursuits for unification, programmability, and performance. To achieve this goal, we present three major components: a kernel programming DSL, a unified runtime, and multiple optimized backends. The DSL adopts the *single-instruction-multiple-threads* (SIMT) programming model for fine controllability and is embedded inside pure modern C++. It aims to simplify the interaction between the host and device code and to enable high-level programming patterns such as polymorphism with *just-in-time* (JIT) code generation and compilation. The unified runtime abstracts the common concepts across different graphics and computing APIs to hide low-level platform-specific details. Atop the low-level interfaces, we also provide wrappers to ease resource management and task scheduling. Behind the unified DSL and runtime, the backends realize the concrete resource management requests and computation tasks on different native APIs. Currently, we have implemented 5 backends for different platforms, namely CUDA, Metal, DirectX, ISPC, and LLVM.

To demonstrate the practicality and efficiency of LuisaRender, we implement a Monte Carlo renderer atop. Our experiments in Sec. 8 show that our renderer achieves significantly faster performance on modern graphics hardware than state-of-the-art open-source renderers, such as PBRT-v4 and Mitsuba 3.

To recap, we contribute a high-performance rendering framework for stream-processing architectures with

(1) a DSL embedded inside modern C++ for kernel programming exploiting JIT code generation and compilation;
(2) a unified runtime layer that hides platform-specific details and provides access to the latest hardware capabilities; and
(3) an optimized Monte Carlo renderer that is 5–11× faster than PBRT-v4 and 4–16× faster than Mitsuba 3 on modern GPUs.

2 RELATED WORK

Stream processors and architectures. Recent advances in computer architectures improve performance via exploiting parallelism from the stream-processing architecture. In this model, data are conceptually arrays flowing through parallel computation units, which can be GPU threads or CPU vector processors. Such processing is usually vectorized as the GPU threads (or CPU SIMD lanes) executing the same program (kernels) on different input data. Vectorization provides better efficiency since it amortizes the cost of fetching and decoding the instructions which are expensive in modern circuit design. This approach is referred to as *single program, multiple data* (SPMD) or *single instruction, multiple threads* (SIMT), and is widely used in existing parallel computation frameworks such as OpenCL, ISPC [Pharr and Mark 2012], and CUDA. We also base LuisaRender upon this programming model for its fine granularity and natural affinity with modern hardware.

Shading languages. Modern graphics APIs are equipped with specific shading languages, such as HLSL, GLSL, Cg [Mark et al. 2003], and the Metal Shading Language [Apple 2021], to support customizable shading effects with high performance. Unlike general-purpose languages such as C++ and Java, shading languages usually have simplified syntax to avoid complicating the compilation and optimization phases. Thus, high-level constructs, such as support for object-oriented and functional programming, are typically missing.

Various efforts have been made by researchers to strengthen the programmability of shading languages. Dating back to the early age of programmable graphics pipelines, the poirneering work Sh [McCool et al. 2002] has already incorporated an embedded DSL in C++ to assemble vertex and fragment shaders at runtime. More recently, He et al. [2017] propose Shader Components to provide

first-class language constructs for shader modularity by encapsulating the shading logics together with the interface parameter blocks. Slang [He et al. 2018] further explores shader specialization and composition, which extends HLSL with generics and interfaces, allowing shader variant generation and management in a more structured and semantic manner. Seitz et al. [2019] construct the Selos shading system atop Terra [DeVito et al. 2013] by exploiting its meta-programming ability. They identify multi-stage programming as an effective approach to reducing the implementation efforts.

Despite the discrepancies in technical choices, these methods share consistent core insights: easy interactions between the host and device and handy language constructs for high-level abstraction. Inspired by them, we design the DSL in LuisaRender to 1) be embedded inside pure C++ to conveniently interact with the host language and runtime; and 2) leverage JIT code generation and compilation to enable multi-stage programming and other high-level patterns with decent performance across platforms.

Rendering systems. Various rendering systems are available nowadays for research and production usages, some adopting mature architecture designs, and some exploring novel technical schemes.

PBRT [Pharr et al. 2016] is an influential open-source renderer, whose recent version adds GPU support. For simplicity and clarity as an educational renderer, it bases the GPU implementation on the straightforward source-level code sharing between CUDA and C++, which depends certain CUDA-exclusive functionalities, such as the unified address space and extended device lambda expressions, hindering the portability toward other platforms. Also, the tagged pointers only allow polymorphic calls to statically known sub-types, limiting the extensibility; and the conventional *ahead-of-time* kernel compilation requires the compiler to generate instructions and allocate registers for every component even if unused. In LuisaRender, we have an extensible support for multiple backends other than CUDA through the unified runtime, so the same code runs across platforms. Moreover, the multi-stage programming ability of our DSL enables dynamic composition and specialization of kernels with runtime information, aggressively inlining scene parameters and eliminating unused instructions and redundant branches.

Mitsuba 2 [Nimier-David et al. 2019] focuses on retargetable and differentiable rendering. It is based on Enoki [Jakob 2019], a computation library for vectorized array programming. While arrays enable a neat and easy-to-retarget programming model, an optimized implementation requires considerable effort, especially on GPUs. Otherwise, the overwhelming memory traffic and thread divergence would lead to severe performance issues, as seen in Mitsuba 2. Jakob et al. [2022a] further propose Dr.Jit, a drop-in replacement of Enoki, to improve the efficiency of array programming. It traces the computation graphs and fuses operations into optimized mega-kernels to eliminate unnecessary memory I/O. Based on Dr.JIT, Mitsuba 3 [Jakob et al. 2022b] achieves a considerable speedup over Mitsuba 2. However, control flows in the array model still rely on the semi-manual management of software execution masks, introducing overhead and redundant computation. Therefore, we opt to stick with the kernel-based explicit SIMT programming model, which receives native support on modern graphics hardware.

Rodent [Pérard-Gayot et al. 2019] is a *renderer generator* built on AnyDSL [Leißa et al. 2018]. Via *partial evaluation*, it takes a particular scene as the static input and specializes the renderer accordingly during compilation. Despite the theoretical grace and optimization potential, the relatively long compilation time (typically several minutes) required for each scene impedes its practicality.

As for production rendering systems, MoonRay [Lee et al. 2017] uses ISPC [Pharr and Mark 2012] to materialize vectorized rendering on CPUs. Manuka [Fascione et al. 2018] leverages JIT shader compilation to fully exploit the SSE and AVX vector instruction sets. Cycles, an open-source path-tracing renderer of the Blender [2022] project, has brought the GPU rendering support to multiple platforms, e.g., Apple Metal, NVIDIA CUDA, and AMD HIP.

Other domain-specific frameworks and languages. Halide [Ragan-Kelley et al. 2012, 2013] is an image-processing DSL. It decouples image processing algorithms from scheduling such as choices of vectorization and tiling, and hence supports easy programming as well as high performance. Afterward, differentiable programming is added to the language [Li et al. 2018] and various algorithms have been proposed for the automatic scheduling of Halide programs [Adams et al. 2019; Anderson et al. 2021; Mullapudi et al. 2016]. Opt [DeVito et al. 2017] and ProxImaL [Heide et al. 2016] are DSLs for efficient image optimization. Aether [Anderson et al. 2017] is a sampling DSL embedded in C++ for automatic probability density calculation from user-written sampling routines.

In the field of physically based simulation, typical frameworks include Taichi [Hu et al. 2019] and Warp [NVIDIA 2022]. Specifically, the Taichi programming language decouples data structures from computation, such that users can easily try various optimized data layouts and access patterns without modifying the core algorithms. Later, Taichi is further extended with automatic differentiation [Hu et al. 2020a] and quantization [Hu et al. 2021].

Beyond graphics, domain-specific frameworks for numeric computation and deep learning are actively proposed and extensively explored. For example, CXYPY [Diamond and Boyd 2016] is a Python-embedded DSL for convex optimization. PyTorch [Paszke et al. 2019], TensorFlow [Abadi et al. 2015], and Jittor [Hu et al. 2020b] represent neural models as computation graphs of basic tensor operations (including addition, multiplication, convolution, pooling, activation functions, etc.) and support automatic differentiation. JAX [Frostig et al. 2018], designed as building blocks for machine learning, employs an array-level programming model and supports JIT compilation and optimization of *pure-and-statically-composed* (PSC) subroutines defined in Python.

Interesting designs and insights of these languages and frameworks also provide valuable inspirations for LuisaRender.

3 SYSTEM DESIGN

Modern renderers are intricate software systems and require a systematic view when designing the underlying computing frameworks. However, the core demands remains simple and clear — effective data representations, friendly yet flexible programming interfaces, and, mostly importantly, efficient computation.

The pursuit of these objectives also motivates our design of LuisaRender. In this section, we will first discuss the design principles and then provide an overview of the resulting architecture.

3.1 Principles

We architect the LuisaRender framework to be unified, easy-to-program, and high-performance.

3.1.1 Unification. The fragmentation of graphics APIs and shading languages adds burdens to the development of cross-platform software: low-level API- and language-specific details require per-platform handling, even though the high-level logic is the same.

We believe such extra efforts are avoidable and that appropriate abstraction layers are the key. For programming languages, compiler frameworks such as LLVM [Lattner and Adve 2004] have proved the effectiveness and elegance of *intermediate representations* to bridge varying front and backends. For graphics APIs, *render hardware interfaces* (RHIs) in modern game engines (e.g., Unreal [Epic Games 2019]) are the core for cross-platform runtime design.

Learning from their success, we introduce similar abstraction techniques into LuisaRender, namely the *abstract syntax trees* (ASTs) and the runtime layer. They decouple the unified frontend interfaces of kernel programming, resource management, and execution scheduling, from the platform-specific backend implementations.

3.1.2 Programmability. Popular shading languages, such as HLSL, are standalone languages with simplified constructs. The lack of support for high-level abstraction such as meta-programming and polymorphism complicates their generation, specialization, compilation, and runtime invocation [He et al. 2018; Seitz et al. 2019].

We resort to an embedded DSL inside modern C++ with JIT code generation and compilation to improve kernel programmability. The rationales are as follows:

- Simplicity. The embeddedness inside pure C++ eliminates the burden of implementing a custom compiler. Instead, it naturally allows reusing the functionalities (e.g., type inference and checking) from the well-established host language and compilers. Also, users will not be bothered to learn yet another shading language and toolset.
- Flexibility. Users can access the host-side data while constructing kernels. Thus, only the necessary kernels need to be dynamically generated and compiled, rather than the numerous combinations of all shader variants.
- Expressiveness. The DSL inherits the rich language constructs from C++, e.g., classes, templates, and lambda expressions. Besides, the host language being a meta-stage of the device-side code, users can programmatically specialize kernels and perform optimizations (e.g., loop unrolling, constant inlining, and de-virtualization), with runtime information and data.

3.1.3 Performance. We expect our framework to exploit modern hardware's capabilities and that applications built atop it could achieve the equivalent performance of those directly on native APIs. This leads to requirements in two dimensions: implementation quality and practical usability.

For implementation, appropriate layering and abstraction granularity offers freedom for high-quality backend implementations. For example, exposing the fine-grained management interfaces and manipulation commands for device resources allows backends to leverage dedicated hardware (e.g., texture units and RT Cores). Also, JIT code generation and compilation is a powerful tool to enable scene-specific dynamic optimizations with runtime information.

For usage, we argue that the viability of high-level abstraction is an assistant to performance rather than a burden, as it unleashes users' productivity and creativity. For example, by extending HLSL with generics and interfaces, Slang [He et al. 2018] enables a modular approach to shader composition and specialization, achieving better performance with fewer implementation efforts than the base language. Meta-programming is also proved effective for high-quality shader generation while relieving the development and maintenance complexity [McCool et al. 2002; Seitz et al. 2019].

Thus, to exploit hardware capabilities, we opt for the SIMT programming model, explicit command-based task submission, and fine-grained resource management. Meanwhile, with high-level DSL constructs and automatic command scheduling, the required programming effort to achieve decent performance is reduced.

3.2 Architecture Overview

Our framework has three major components (shown in Fig. 1): an embedded DSL for kernel programming, a unified runtime for resource management and execution scheduling, and multiple backends implementing the low-level platform-specific functionalities. A short example with our framework is shown in Listing 1.

3.2.1 Embedded DSL. The DSL in our system provides a unified approach to authoring kernels, i.e., programmable computation tasks on the device. Distinct from typical graphics APIs that use standalone shading languages for device code, our system unifies the authoring of both the host-side logic and device-side kernels into the same language, i.e., modern C++.

The implementation purely relies on the C++ language itself, without any custom preprocessing pass or compiler extension. We exploit meta-programming techniques to simulate the syntax, and function/operator overloading to dynamically trace the user-defined kernels. ASTs are constructed during the tracing as an intermediate representation and later handed over to the backends for generating concrete, platform-dependent shader source code.

3.2.2 Unified runtime. Likewise the RHIs in game engines, we introduce an abstract runtime layer to re-unify the fragmented graphics APIs across platforms. It extracts the common concepts and constructs shared by the backend APIs and plays the bridging role between the high-level frontend interfaces and the low-level backend implementations.

On the low-level end, unification involves two parts: 1) the *abstract device interfaces* for resource management, e.g., buffer allocation and de-allocation; and 2) the *abstract commands* as intermediate descriptors for device-side computation and resource manipulation, such as launching kernels and building acceleration structures.

On the programming interfaces for users, we provide high-level resource wrappers to ease programming and eliminate boilerplate code. They are strongly and statically typed modern C++ objects, which not only simplify the generation of commands via convenient

```cpp
// initialize the device
auto device = context.create_device("cuda");

// define the rendering kernel
Kernel2D kernel = [&](ImageFloat image) {
  auto pos = dispatch_id().xy();
  auto color = sin(make_float2(p)) * .5f + .5f;
  image.write(pos, make_float4(color, 1.f, 1.f));
};

// create resources on the device and on the host
auto size = make_uint2(1024u);
auto render = device.compile(kernel);
auto image = device.create_image<float>(BYTE4, size);
auto host_image = std::vector<float4>(size.x * size.y);

// create a stream for submitting tasks
auto stream = device.create_stream();

// dispatch tasks and wait for completion
stream << render(image).dispatch(size)
       << image.copy_to(host_image.data())
       << synchronize();
```

Listing 1. A short example program with our framework.

member methods but also support close interaction with the DSL. Moreover, with the resource usage information in kernels and commands, the runtime automatically probes the dependencies between commands and re-schedules them to improve hardware utilization.

3.2.3 Backend implementations. The backends are the final realizers of computation. They generate concrete shader sources from the ASTs and compile them into native shaders. They implement the virtual device interfaces with low-level platform-dependent API calls and translate the intermediate command representations into native kernel launches and command dispatches.

Currently, we have 5 backends, including 3 GPU backends based on CUDA, Metal, and DirectX, respectively, a scalar CPU backend on LLVM, and a vectorized CPU backend on ISPC.

4 EMBEDDED DOMAIN-SPECIFIC LANGUAGE

To enable a unified experience of kernel authoring, we implement an embedded domain-specific language in C++, which is designed to maximize familiarity with native C++ and other popular shading languages. Meanwhile, embedding the language in pure C++ greatly simplifies the compilation process and allows a seamless interaction with other parts of the system as well as powerful, high-level programming patterns hardly seen in previous work.

In this section, we will discuss the basic components and usages of the DSL, introduce the techniques behind the scene to materialize and drive the syntax and compilation, and describe some high-level abstraction patterns to showcase its rich expressiveness.

4.1 Basic Components and Usages

The embedded language is designed to enable a fluent experience authoring kernels and blur the boundaries with the host code. To achieve this, we provide language constructs similar to the original C++, honoring its features and idioms.

Types and variables. As in most imperative programming languages, the basic building blocks of kernels in our DSL are variables, which

```cpp
template<typename T>
class Var { /*...*/ };

/* aliases for commonly used instantiations */
using Int  = Var<int>;
using Int2 = Var<int2>;
using Int3 = Var<int3>;
using Int4 = Var<int4>;
/* ... */

/* aliases for runtime resources */
using BufferInt = Var<Buffer<int>>;
using ImageInt  = Var<Image<int>>;
/* ... */
```

Listing 2. Types of the variables in the DSL.

are typed as wrapper templates upon the scalar types (i.e., `int`, `uint`, `float`, and `bool`) and the derived vector and matrix types (i.e., `int2`, `bool3`, `float3x3`, etc.) resembling those in popular shading languages, as well as the runtime resources aforementioned in Sec. 5.2.

Listing 2 displays the definition of the `Var<T>` template, which is the core basis for all the behind-scene techniques of AST tracing and recording that are revealed in Sec. 4.2. Unlike vanilla C++ variables, the class is substantially designed as proxies to kernel variables, whose operators and methods are overloaded to only *record* the operations rather than actually perform the calculation. Also, for users' convenience, we pre-define aliases to the most commonly used instantiations, which are in a capitalized naming convention, clearly distinguishable from the native types.

Definitions of DSL variables are the same as ordinary objects in C++: the name of the variable preceded by its type, optionally followed by an initializer. Leveraging user-defined guides for the *class template argument deduction* (CTAD) in C++, we can omit explicit specifications of the template arguments of the `Var` class and let the compiler automatically deduce them from the initializers, analogous to the `auto` keyword. Besides, implemented in pure C++, the DSL types are naturally allowed as class members, function parameters, and return types, even in a mixture with other C++ types. The following listing shows some of the use cases.

```cpp
/* defining local variables as natively in C++ */
Int   a;       // an int variable without initializers
Float b = 1.f; // a float variable initialized to 1.0f
Int   c = a;   // initialized with other variables, or
Bool2 d{true, false};      // with an initializer list

/* defining local variables with type deduction */
Var  e = 256u; // deduced to UInt by user-defined CTAD
auto f = Var{256u}; // alternative style 1: ad hoc Var
auto g = def(256u); // alternative style 2: def helper

/* in a function signature mixed with ordinary types */
Bool3 test(BufferFloat b, int static_index) { /*...*/ }

/* as class members, mixable with ordinary types */
class Bar { std::string name; int id; Float3 v; };
```

Expressions and statements. Program logic is assembled by transforming and transfering data held in variables. We provide arithmetic, relational, and assignment operators of DSL variables for basic computation in kernels. Exploiting operator overloading and type traits, we furnish the DSL with almost the same interfaces as the

```
$if (cond) { /*...*/ };
$if (cond) { /*...*/ } $else { /*...*/ };
$if (cond) { /*...*/ } $elif (cond2) { /*...*/ };
$while (cond) { /*...*/ };
$for (variable, n) { /*...*/ };
$for (variable, begin, end) { /*...*/ };
$for (variable, begin, end, step) { /*...*/ };
$loop { /*...*/ }; // infinite loop, unless $break'ed
$switch (variable) {
  $case (value) { /*...*/ };
  $default { /*...*/ };
};
$break; $continue;
```

Listing 3. Special macros for control flows in the DSL.

original C++ syntax, together with static type checking, inference, and conversion support, even when mixed with native types.

```
auto a = def(0u);           // Var<uint> defined in DSL
auto b = def(1u);           // Var<uint> defined in DSL

/* operators, assignments, and type inference */
auto c = a + b;       // operator+: (uint, uint) -> uint
auto d = a < b;       // operator<: (uint, uint) -> bool
b = a - c * 3u;       // operator- and *, and assignment

/* static type check and conversion */
auto u = 1 + c;     // literal int(1) converted to uint
// float3(1.f) + u => compile-time error: float3 + uint
```

Control flows. In addition to arithmetic and assignment statements, control flows also play important roles in program construction. Unfortunately, they are not overloadable in C++, nor can we detect and extract them within the language itself. Therefore, in the DSL, we opt for special macros to imitate them (see Listing 3), which are prefixed by the `$` sign, a rarely used but valid character in identifiers, to be told apart from the native C++ counterparts. We will describe their implementation in Sec. 4.2.

Kernels and callable functions. LuisaRender supports two categories of device functions, namely, 1) `Kernel`s (1D, 2D, or 3D), which are entries to the parallelized computation tasks on the device; and 2) `Callable`s that are function objects invocable from kernels or other callables. Both kinds are template classes that are constructible from C++ functions or function objects including lambda expressions. Again, leveraging CTAD guides, the template parameters can optionally be omitted and deduced by the C++ compiler. At the code generation stage in the backend, they are mapped and translated to corresponding function entities in the target shading/programming language, e.g., `Kernel`s into `__global__` functions and `Callable`s into `__device__` functions in CUDA.

It is also worth mentioning that, since our DSL is itself valid C++, ordinary C++ functions and function objects, including class member functions and lambda expressions, are naturally available too. When used together with the DSL, they act like macros that are directly expended into the ASTs, without generating function entities as `Kernel`s and `Callable`s. In other words, they are *meta-stages* that control the assembly of the kernel ASTs, which can be extremely powerful for composing higher-order abstraction patterns as will be discussed in Sec. 4.3.

Listing 4 shows example callable and kernel functions written in our DSL, which, in combination, write gradient color to an image (i.e., 2D texture) in sRGB encoding. By exploiting type-traits and defining deduction guides, the template arguments for types `Callable` and `Kernel2D` are automatically inferred from the signatures of the lambdas, e.g., deduced to be `Callable<float3(float3)>` and `Kernel2D<ImageView<float>>`, respectively.

```
Callable to_srgb = [](Float3 x) {
  $if (x <= 0.00031308f) {
    x = 12.92f * x;
  } $else {
    x = 1.055f * pow(x, 1.f / 2.4f) - .055f;
  };
  return x;
};
Kernel2D fill = [&](ImageFloat image) {
  auto coord = dispatch_id().xy();
  auto size = make_float2(dispatch_size().xy());
  auto rg = make_float2(coord) / size;
  // invoke the callable
  auto srgb = to_srgb(make_float3(rg, 1.f));
  image.write(coord, make_float4(srgb, 1.f));
};
```

Listing 4. An example code snippet using our DSL.

Note that for `Kernel`s and `Callable`s to correctly trace the ASTs, parameters of the underlying definition functions must be wrapped in `Var<T>`, or equivalently, use aliases such as `Float3` and `ImageFloat`.

Built-in functions. Besides user-defined functions, we also provide a rich library of built-in DSL functions. They are typically intrinsic functions that are not possible (or at least not efficient) to be implemented in user code and hence must be supplied by the framework. In LuisaRender, built-in functions include

- Thread coordinate and launch configuration queries, including `block_id`, `thread_id`, `dispatch_size`, and `dispatch_id`;
- Mathematical routines, such as `max`, `abs`, `sin`, `pow`, and `sqrt`;
- Resource accessing and modification methods, such as texture sampling, buffer read/write, and ray intersection;
- Variable construction and type conversion, e.g., `def<T>` for making variable copies, `make_int3` for creating 3D integer vectors, and `as<T>` for bitwise type casting; and
- Optimization hints for backend compilers, which currently consist of `assume` and `unreachable`.

We exploit *concepts* in C++20 to constrain the signatures of built-in functions such that the compiler would not confuse them with the host functions. In the DSL, invocations to them will record `CallExpr`s with special tags in the AST.

During code generation, the backend maps the built-in invocation nodes to platform-specific code. Simple functions like `thread_id` and `abs` are directly mapped to shader intrinsics, possibly backed by hardware instructions, while the more complicated ones, such as ray intersection, might be forwarded to pre-defined functions or even external libraries. Also, on platforms without native support for some functionalities, we have to simulate the corresponding semantics with software implementations. For example, on Metal, we simulate atomic floating-point operations through the *compare-and-swap* (CAS) operation on atomic integers.

```
/* option 1:
 * use DSL members in device-only classes */
class BSDFSample {
public:
  Float3 f;
  Float3 w;
  Float pdf;
  auto valid() const { return pdf != 0.f; }
};

/* option 2:
 * reflect host-side structures for device usage */
struct BSDFSample { // host-side ordinary C++ structure
  float3 f;
  float3 w;
  float pdf;
};

LUISA_STRUCT(BSDFSample, f, w, pdf) { // DSL reflection
  auto valid() const { return pdf != 0.f; }// extension
};
// briefly, the macro expends to
// template<>
// class Var<BSDFSample> {
// public:
//   Var<decltype(f)> f;
//   Var<decltype(w)> w;
//   Var<decltype(pdf)> pdf;
//   auto operator->() {// for extension methods
//     return (VarExtension<BSDFSample> *)this;
//   }
// };
// template<>
// struct VarExtension<BSDFSample> : Var<BSDFSample> {
//   auto valid() const { return pdf != 0.f; }
// };

// after the registration, the host-side structure is
// now available in DSL for variables and resources
Buffer<BSDFSample> sample_bufer = /* ... */;
Var<BSDFSample> sample = sample_buffer.read(0);
Bool is_valid = sample->valid();
```

Listing 5. Two ways in our DSL to incorporate C++ structures and classes. The first is to use DSL variables as class or structure members, and the second is to reflect and register existing C++ structures.

Structures and classes. In C++, structures and classes are important constructs to organize data and express logic. Since variables in our DSL are intrinsically C++ objects, they are naturally allowed as structure or class members and used inside device functions. In this case, structures and classes act like DSL variable packs, usages of which are instantly inlined without any entities generated in the backend shading language.

However, to conveniently exchange data between the host and device side through buffers, we usually also want the ability to reflect and register existing C++ structures. We provide a macro, `LUISA_STRUCT`, to fulfill this need, which leverages type traits to automatically inspect the types of the members and initialize the corresponding `Var<T>` template for DSL compatibility. Also, resembling the object-oriented fashion, the macro supports extending the original structures with device-side methods.

Listing 5 gives examples of incorporating C++ classes and structures in the aforementioned ways.

4.2 Syntax Tree Recording

The DSL traces and records the *abstract syntax trees* (ASTs) of `Kernel`s and `Callable`s through extensive usage of macro and template meta-programming. The ASTs, together with the captured runtime information, are later forwarded to backends for code generation, compilation, and pipeline creation.

Conceptually, the process can be interpreted as a *dynamic* implementation of *multi-stage programming* [Taha 2004]: like what templates and macros are to classes and functions, our DSL is a meta-programming stage to the backend shading language, which controls the assembly of the generated code. That said, the dynamic information (that is available only at runtime for the host C++ language), is known "statically" concerning shaders and thus integratable into their construction and compilation.

Plus, the DSL is embedded natively in modern C++, requiring no special pre- or post-processing of the source code, nor customized modifications or extensions to the compilers. Thus, all C++ language constructs as well as programming patterns, idioms, and paradigms, either object-oriented, procedural, or functional, are seamlessly compatible with our system, putting atop another layer of programmability at no overhead regarding the backend device.

Tracing of the syntax tree is through the proxy objects of the `Var<T>` template class. Operators, as well as built-in functions (e.g., `pow`), are all overloaded for the proxy type `Var<T>`, so that mathematical operators on them or function calls to them do not compute the result, but rather add nodes into the AST. It is also worth noting that, by using type-traits and concepts, the DSL is statically and almost strongly typed, such that type-level errors are promptly reported during the compilation of the host C++ language, reducing the effort required to debug at runtime.

Below we explain how statements and function definitions are recorded, and how control flows are handled.

Recording of statements. The pseudocode in Listing 6 is an example showing how a single statement in C++ (i.e., `auto y = x * 12.92f;`) is traced and recorded into the AST (suppose `x` is a `Var<float>`).

However, control flows (e.g., `if`, `while`, and `break`) are not overloadable. Instead, we use special macros, such as `$if`, `$while`, and `$break`, to imitate these statements, by intruding into the syntax of lambda expressions. For example, the definition of `$if`-related constructs are shown in Listing 7, followed by the expansion of Line 2 to 6 in Listing 4, where the `IfStmtBuilder` has overloaded `operator%` and `operator/` for recording the *true* and *false* branches, respectively, by invoking the corresponding stitched lambda expressions.

Should our readers have noticed, this also explains why there must be braces around the body statements and semicolons after the closing braces, as they are parts of the lambda-expression syntax.

Recording of function definitions. Types of the device functions, `Kernel` and `Callable`, are in fact template classes, whose constructors drive the recording of the passed-in functions. Briefly, on calling the constructor, the following steps are taken to build the AST:

(1) a new `FunctionBuilder` that stores the AST is created and pushed to a global stack to open a new function scope;
(2) parameter types of the passed-in function are automatically detected to proxy `Var<T>` argument objects;

```
// C++ statement: auto y = x * 12.92f;
// wrap the left-hand-side operand for operator*
Expr<float> lhs{x.expression()};
// wrap the right-hand-side literal for operator*
Expr<float> rhs{builder->literal(
  Type::of<float>(), 12.92f)};
// create a BinaryOp node
const Expression *binary = builder->binary(
  Type::of<float>(), BinaryOp::Tag::MUL,
  lhs.expression(), rhs.expression());
// construct variable y from the result
Var<float> y{builder->local(Type::of<float>())};
builder->assign(y.expression(), binary);
```

Listing 6. Recording process of a DSL statement.

```
#define $if(...)                        \
  detail::IfStmtBuilder{__VA_ARGS__} \
  % [&]()
#define $else                           \
  / [&]()

// expansion process:
// create a temporary if-statement node builder
detail::IfStmtBuilder{x <= 0.00031308f}
// operator%: records the true branch
% [&]() { x = 12.92f * x; }
// operator/: records the false branch
/ [&]() { x = 1.055f * pow(x, 1.f / 2.4f) - .055f; };
```

Listing 7. Macro definition of the `$if`-related constructs and the example expanding process of Line 2 to 6 in Listing 4.

(3) the passed-in function is invoked with the proxy arguments and its execution drives the tracing of the AST; and
(4) at the return of the definition function, the previously opened function scope is closed and the syntax tree stored in the `FunctionBuilder` is handed over to the newly constructed `Kernel` or `Callable` objects.

For example, the definition (Line 1 to 8) of the callable `to_srgb` in Listing 4 basically expands to the following simplified pseudocode (suppose the definition function, i.e., the lambda expression in this example, is passed to the constructor as `to_srgb_impl`):

```
// create a callable builder
auto builder = FunctionBuilder::create(Tag::CALLABLE);
// push the new function scope
FunctionBuilder::push(builder);
// create a expression node for the the argument
auto arg0 = builder->argument(Type::of<float3>());
// wrap the node into a proxy object
Var<float3> v0{arg0};
// invoke the definition function, all DSL operations
// in to_srgb_impl are traced and recorded in the AST
auto ret = to_srgb_impl(std::move(v0));
// record the return statement (if any)
builder->return_(ret.expression());
// pop function scope
FunctionBuilder::pop(builder);
// construct the callable object with the builder
Callable<float3(float3)> to_srgb;
to_srgb._builder = std::move(builder);
```

Recording of function calls. There are two categories of functions supported in the DSL: built-in functions and user-defined ones. Both are statically typed and calls to them add `CallExpr` nodes in the ASTs.

```
template<typename T1, typename T2>
inline auto cosine_sample_hemisphere(T1 u1, T2 u2) {
  auto r = sqrt(u1);
  auto phi = 2.f * pi * u2;
  auto x = r * cos(phi);
  auto y = r * sin(phi);
  auto z = sqrt(1.f - u1);
  return make_float3(x, y, z);
};
```

Listing 8. A generic function compatible with both the host C++ language and the DSL in our system.

```
inline auto tea(UInt s, UInt v0, UInt v1) {
  for (auto i = 0; i < 4; i++) {
    s += 0x9e3779b9u;
    v0 += ((v1 << 4) + 0xa341316cu) ^ (v1 + s) ^
          ((v1 >> 5u) + 0xc8013ea4u);
    v1 += ((v0 << 4) + 0xad90777du) ^ (v0 + s) ^
          ((v0 >> 5u) + 0x7e95761eu);
  }
  return v0;
};
```

Listing 9. User-controlled loop unrolling of the *tiny encryption algorithm* [Zafar et al. 2010] hash function, easily achieved in our DSL by repeatedly (4 times in the example) recording the statements within a host loop.

For example, `pow` on the 5th line of Listing 4 is a built-in function and `to_srgb` itself is a custom callable function, which is invoked on the 14th line. Note that callables are invocable both in kernels and other callables, while kernels are only allowed to be dispatched and executed through command streams after compilation.

Besides, ordinary C++ functions, as mentioned before, are also available for use in our DSL. Different from kernels and callables, they are directly expended and inlined into the ASTs without creating device function entities and can be used for higher-order abstraction patterns as will be discussed in the next subsection.

4.3 Advanced Features

In this part, we will discuss the advanced abilities inherited from the host C++ languages, as well as those possibilities novelly discovered and developed with our system.

Generic functions compatible with both the host code and DSL. Templates are primarily designed for generic programming. Our DSL, natively implemented in C++, is compatible with templates.

Listing 8 showcases the inherited generic programming ability, which defines a cosine-weighted hemisphere sampling routine compatible with both the host C++ language and DSL. More sophisticated cases, e.g., template classes, are also available in our system.

Fine-grained control over code generation. As discussed in Sec. 4, the DSL in our system can be interpreted as a wrapper of the AST building functionalities. Therefore, it naturally allows users to have fine-grained control over the code generation process, if preferable. For example, user-controlled loop unrolling is easily achievable by repeatedly recording the AST nodes of the corresponding statements several times as in Listing 9.

```
// defined in plug-in: tonemapping_aces.dll
Float3 apply(Float3 in) { /*...*/ }
// defined in plug-in: tonmapping_filmic.dll
Float3 apply(Float3 in) { /*...*/ }
// defined in plug-in: tonemapping_uncharted2.dll
Float3 apply(Float3 in) { /*...*/ }
/* ... */

// a polymorphic tonemapping shader creator
// parameterized by a higher-order function
auto create_kernel(function<Float3(Float3)> op) {
  Kernel2D kernel = [&op](ImageFloat image) {
    auto p = dispatch_id().xy();
    auto color = image.read(p);
    // op(): a host-side dynamic call, expanding the
    // polymorphic logic into the shader, which is
    // effectively de-virtualized on the device side
    auto mapped = op(color.xyz());
    image.write(p, make_float4(mapped, color.w));
  };
  return kernel;
}

// at runtime, load a tonemapping plug-in dynamically
auto tm_plugin = load_module(/*...*/);
// now use the dynamically loaded op to create kernels
auto tm_kernel = create_kernel(tm_plugin.get("apply"));
/* ... */
```

Listing 10. Dynamic polymorphic host code enabling construction of shader variants at runtime.

4.4 Dynamic Polymorphism

A rendering system consists of various categories of textures, surface materials, light sources, etc., which are ideally handled using polymorphism. However, language support for dynamic polymorphism, such as virtual functions and pattern matching, are limited or even absent in the GPU world. The conventional approach to working around is to generate numerous *shader variants* by enumerating all possible combinations from code snippets or high-level components [He et al. 2017] ahead of time and selecting them at runtime. However, such a solution wastes not only the compilation and packaging time of unnecessary variants but also the time and memory to load and hold them at runtime. Another technique, as extensively explored in PBRT-v4 [Pharr et al. 2016], is to use *tagged pointers* to dynamically dispatch polymorphic method calls, which, unfortunately, requires support for templates in the shading language and full knowledge of all subtypes (i.e., derived classes), thus greatly hurting the portability and extensibility. Moreover, statically compiled without the chance to prune unused subtypes, the tagged pointer approach is unable to alleviate register pressure and always has a constant branching overhead for each polymorphic call.

Our system, taking a different approach, leverages the embedded DSL and just-in-time compilation to support polymorphism, achieving both flexibility and performance. Specifically, we implement two types of dynamic polymorphisms. The first is achieved through *de-virtualization*, which provides dynamic polymorphism on host code while reduces to static polymorphism on device code. The second is achieved through *dynamic tagged dispatching*, which provides true dynamic polymorphism on device code.

De-virtualized host-side polymorphism. As an elegant and flexible alternative to shader variants, we incorporate the subtyping and dynamic dispatching mechanisms in the host language to assemble shaders at runtime with different components. In this case, the DSL acts like dynamic macros, extracting the polymorphic logic into the different ASTs. Listing 10 gives a functional-style example reflecting this ability, one in a functional programming style with high-order functions and the other in an object-oriented fashion with virtual, dynamically dispatched methods.

```
class BRDF {
  virtual Eval eval(Float3 wo, Float3 wi) = 0;
};
class Lambertian : public BRDF {
  Eval eval(Float3 wo, Float3 wi) override { /*...*/ }
};
class Microfacet : public BRDF {
  Eval eval(Float3 wo, Float3 wi) override { /*...*/ }
};
class BRDFEvaluator {
private:
  Polymorphic<BRDF> _f;
public:
  void do_registration() {
    auto tag1 = _f.create<Lambertian>(); // tag1 == 0
    auto tag2 = _f.create<Microfacet>(); // tag2 == 1
    // register other BRDFs...
  }
  auto evaluate(Hit hit, Float3 wo, Float3 wi) {
    Eval eval;
    _f->dispatch(hit->brdf_tag(), [&](auto f) {
      eval = f->eval(wo, wi);
    });
    // equivalently expands to
    // $switch (hit->brdf_tag()) {
    //   /* calling Lambertian::eval() */
    //   $case(0) { eval = _f[0]->eval(wo, wi); };
    //   /* calling Microfacet::eval() */
    //   $case(1) { eval = _f[1]->eval(wo, wi); };
    //   ...
    // };
    return eval;
  }
};
```

Listing 11. Dynamic polymorphism achieved on device with the templated `Polymorphic<T>` construct.

Such dynamism is engaged in the AST construction stage and wiped out in the device code, which, from the perspective of compiler optimization, resembles a *de-virtualization pass* that aggressively eliminates redundant indirect calls, branches, and register allocation, effectively improving the hardware occupancy and utilization.

Device-side dynamic dispatches. We provide a more efficient and agile replacement to virtual functions and tagged pointers [Pharr et al. 2016] when device-side dynamically polymorphic calls are inevitable, by utilizing the meta-programming ability to dynamically generate tagged dispatches in the shader ASTs at runtime. In this case, we offer a helper template class named `Polymorphic`, which, as shown in Listing 11, automatically assigns tags to each subtype implementation or object instance and constructs the tagged dispatch logic on demand by generating `switch` statements in the AST, through loop unrolling (Sec. 4.3).

Compared to function pointers and virtual functions, this approach does not require high-level object-oriented or functional features in the shading languages and is generally more efficient without the overhead of indirect function calls. Meanwhile, the

method is more flexible than tagged pointers, in that it constructs the dispatch logic at runtime without the need to know all subtypes statically. More importantly, it allows generating code only for used subtypes, which potentially reduces code size and register usage, providing better performance.

4.5 Interactions with the Runtime

The DSL is closely tied to the runtime: device resources are easily captured, tracked, and bound; resource usage information flows through the runtime to the backends; DSL function interfaces are transformed from kernels to compiled shaders, etc. Such interactions between the DSL and the runtime make it possible to achieve great performance while maintaining an extensible and easy-to-use API.

Resource capturing. While binding arguments by name or index for resources is a common choice in most compute APIs, it is cumbersome and error prone, and mistakes are reported only at runtime or even unpredictably lead to undefined behaviors. In contrast, our DSL allows users to use resources via capturing. Suppose we have created a 2D image as follows, by capturing it in the lambda and calling to its `write()` method, a kernel argument will be automatically declared and bound to this device resource, without the need to put it in the function signature.

```
auto image = device.create_image<float>(/* ... */);
Kernel2D fill = [&] {
  auto coord = dispatch_id().xy();
  auto rg = make_float2(coord) /
            make_float2(dispatch_size().xy());
  auto srgb = to_srgb(make_float3(rg, 1.f));
  image.write(coord, make_float4(srgb, 1.f));
};
```

As will be demonstrated in Sec. 7, this recording-by-capturing feature is extremely useful, which provides applications with full freedom to hold distinct resources while maintaining the same interfaces, thus blurring the boundary between host and device code and enabling extra patterns of polymorphism.

Interface transformation and shader dispatching from the host. Leveraging template meta-programming, the signature of a kernel (i.e., parameters with type `Var<T>`) is converted into a statically typed shader interface. Any invocation through this interface will then automatically type-check the resources and create a dispatch command recording the binding information.

```
Kernel2D kernel = [](Var<Image<float>>, Var<uint2>) {
  /* ... */
};
```

For example, suppose we have the kernel above, compiling it yields a shader with an interface-transformed `operator()`. Invocations to this interface construct shader dispatch commands, which are later engaged in command buffers, committed through the stream, and executed by the backend device.

```
Kernel2D kernel = [](Var<Image<float>>, Var<uint2>) {
  /* ... */
};
auto shader = device.compile(kernel);
// Now shader has a transformed interface:
// Shader2D<Image<float>, uint2>
//   ::operator()(ImageView<float>, uint2);

// invoke the transformed interface
auto cmd = shader(image, size).dispatch(512, 256);
// which expends to
// auto cmd = ShaderDispatchCommand::create(
//     shader.handle(), shader.function());
// cmd->encode_texture(image.handle(), image.level());
// cmd->encode_uniform(&size, sizeof(uint2));
// cmd->set_dispatch_size(512, 256, 1);
```

The interface transformation feature is fully compatible with the aforementioned resource capturing capability and handles the captured resources in a consistent way — they are simply implicitly bound items in the shader dispatch commands.

Usage tracking for resources. With full knowledge of the kernels and their ASTs, we can analyze the usages (e.g., read-only, write-only, and read-write) of any resources in the kernels, which are useful for further optimizations. For example, if we found a buffer argument is never written to in a kernel, then the buffer view bound to the corresponding shader is marked read-only, hence the opportunity to execute the shader dispatch command in concurrency with other read-only commands. More details are discussed in Sec. 5.3.

5 UNIFIED RUNTIME LAYER

To make our system cross-platform, we introduce a unified runtime layer that defines the common abstract interfaces for different backends and provide convenient wrappers for users. One major functionality of the layer is the management of *resources*, which provides interfaces and wrappers to create, destroy, organize, and access data for computation. Another important part is to support the description, submission, and scheduling of *commands*, such as transferring data between the device and the host, or invoking kernels written in our DSL. Besides, the runtime layer also performs common optimizations based on runtime information.

5.1 Abstract Device Interfaces

Analogous to RHIs in game engines, the abstract device interfaces in LuisaRender are a set of thin routines that extract common concepts and patterns across different backends, so that logic built upon them is liberated from platform-dependent details.

Our interface class is shown in Listing 12. It declares virtual methods to communicate with the backends, including creating, destroying and manipulating device resources, dispatching commands (such as invoking kernel computation), and methods for synchronization. New backends can be easily added and dynamically loaded as plugins, as long as they inherit from this class and implement its virtual methods. For flexibility in the backend implementations, we use *handles*, each as a 64-bit unsigned integer, to represent the device resources and stream management components.

5.2 Resources and Their Wrappers

Device Resources. As we have mentioned, one of the major parts of the runtime layer is resource management. We categorize device resources into the following classes (some illustrated in Fig. 2):

- Buffers, which are linear memory ranges on the device for structured data storage;

```
class Interface {
public:
  // methods for creating, destroying, and managing
  // resources, including buffers, textures, meshes,
  // shaders, bindless resource arrays, acceleration
  // structures, streams, and synchronization events.
  virtual u64  create_buffer(size_t size)          = 0;
  virtual void destroy_buffer(u64 handle)          = 0;
  /* ... */

  // methods for dispatching commands and shaders
  virtual void dispatch(u64 stream, CommandList)   = 0;
  virtual void dispatch(u64 stream, Callback)      = 0;

  // methods for synchronization
  virtual void synchronize_stream(u64 handle)      = 0;
  virtual void signal_event(u64 event, u64 stream) = 0;
  virtual void wait_event(u64 event, u64 stream)   = 0;
  virtual void synchronize_event(u64 handle)       = 0;

  // miscellaneous methods, e.g., feature query
  /* ... */
};
```

Listing 12. The abstract device interface class.

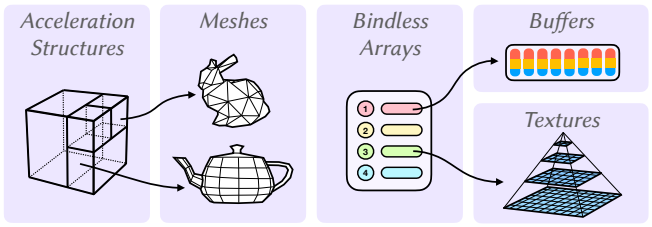


Fig. 2. Various hardware resources, including accelaration structures for ray-mesh intersection tests, buffers for storing structured data, 2D/3D textures for sampling and bindless arrays as indices of buffers and textures.

- Textures, which are 2D images or 3D volumes of scalars or vectors, typically with optimized layouts and dedicated hardware for caching, sampling, and format conversion;
- Bindless arrays, which provide slots for references to buffers and textures, helpful for reducing the overhead and bypassing the limitations of binding shader parameters;
- Meshes and acceleration structures for high-performance ray intersection tests, with hardware-accelerated if available (e.g., on graphics cards that feature RT-Cores);
- Shaders, i.e., kernels that are compiled into pipeline state objects for general-purpose computation on device; and
- Streams and events for command submission and intra- or inter-stream synchronization.

They are common concepts supported by modern graphics and computing APIs, though usually referred to by different terminologies.

Readers might notice that these concepts are often absent from other research work, and doubt the necessity of this "complication" instead of using a single, seemingly more elegant abstraction, e.g., fields in Taichi [Hu et al. 2019] and arrays in Enoki [Jakob 2019] and Dr.JIT [Jakob et al. 2022a]. However, the various workloads in an intricate rendering system often benefit from optimized data layouts and specialized hardware units, such as the texture units

```
template<typename T>
class Buffer final : public Resource {
  /* ... */
public:
  // for DSL usage
  template<typename I>
  auto read(I &&i) const noexcept {
    return Expr{*this}.read(std::forward<I>(i));
  }
  // for convenient command generation
  auto copy_from(BufferView<T> src) const {
    /* some lines of bounds checking */
    return BufferCopyCommand::create(
      src.handle(), src.offset_bytes(),
      handle(), 0u /* dst_offset */, size_bytes());
  }
  /* other methods */
};
```

Listing 13. The wrapper for buffers and their views.

for efficient image sampling and hardware-accelerated BVHs and ray intersection units for fast ray tracing. Therefore, targeting high performance, we decide to leave the freedom of choice to our users. We believe the clear semantics of the resources will ease, not harden, the development of graphics applications.

Resource Wrappers. The abstract device interface (Listing 12) provides a unified layer for low-level backend functionalities. To hide excessive details from users, we implement wrapper classes for the aforementioned resources on top of this layer. They conform to the resource-acquisition-is-initialization (RAII) idiom in C++ to simplify the lifetime and ownership management, with member methods for resource usage and management in the DSL and host code.

For example, in Listing 13, the `Buffer` template class supplies a `read` method, invocations of which are forwarded to the DSL and the function builder, and they automatically register argument bindings as well as typed `BUFFER_READ` instructions in the AST, with a DSL variable representing the result returned (see Sec. 4 for details on how this works). Calling the wrapper's `copy_from` method with another buffer creates a `BufferCopy` command that can be later inserted in a stream and submitted to the backend device.

5.3 Command Encoding and Submission

Command-based model. Conceptually, *commands* are description units of atomic computation tasks, such as transferring data between the device and host, or from one resource to another; building meshes and acceleration structures; populating or updating bindless arrays; and most importantly, launching shaders. Commands are organized into *command buffers* and then submitted to *streams* which are essentially queues forwarding commands to the backend devices in a logically first-in-first-out (FIFO) manner. Since multiple streams run concurrently, users may require synchronizations between them or with respect to the host via *events*, similar to condition variables that ensure ordering across threads.

With explicit command buffers and fully asynchronous submission, this command-based model clarifies the execution boundaries between tasks and brings optimization opportunities to scheduling, well matching our design principle of performance.

```
/* example usages of streams and command buffers */
auto command_buffer = stream.command_buffer();
command_buffer
  << raytrace_shader(framebuffer, accel, resolution)
      .dispatch(resolution)
  << accumulate_shader(accum_image, framebuffer)
      .dispatch(resolution)
  << hdr2ldr_shader(accum_image, ldr_image)
      .dispatch(resolution)
  << ldr_image.copy_to(host_image.data())
  << commit();
```

Listing 14. A use case of command buffers in a simple path tracer.

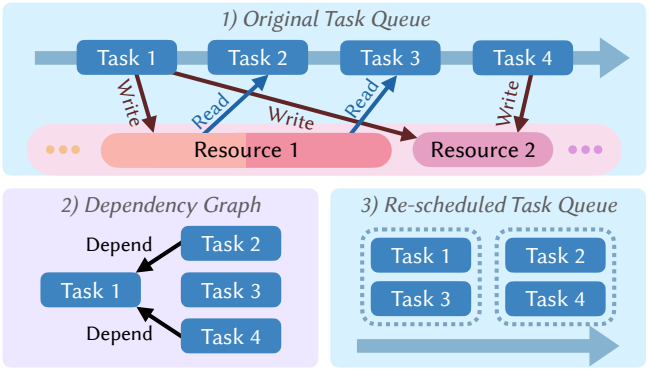


Fig. 3. Dependencies between commands are detected using resource tracking. Although task 1 and task 3 seemingly occupy the same resource, they are independent because they use non-overlapping regions of the resource. Commands are then re-scheduled into sub-lists to exploit concurrency.

Command buffer and stream. Command buffers are designed for low-overhead asynchronous command submission, likewise modern graphics APIs. Users can encode and stage commands in a list and submit them to the stream at once, which reduces the frequencies of state preparation and switching. Command buffers are created from streams and have an overloaded `operator<<` for a clean syntax to append commands and a `commit` method for submitting gathered commands to their creator streams. Listing 14 shows a use case of rendering and downloading a frame in a mega-kernel path tracer. Togther with the command creation methods in resource wrappers, the overloaded `operator<<` makes the process succinct.

The stream submits the command list to the backend through the interface defined in Listing 12. The backend then fetches the serialized commands from the list, deserializes and translates them into native commands and API calls, and eventually hands them over to the compute devices for execution. The seemingly redundant serialization and deserialization processes play important roles in abstracting away the disparities across backends. Besides, since the commands in a stream are asynchronously executed to the host and multiple streams run in parallel to each other, we provide events for stream-to-stream and stream-to-host synchronizations.

Resource tracking and command scheduling. The usage of command buffers, together with streams, eliminates the overhead of state bookkeeping and frequent API calls, allows parallel command encoding, and enables a fully asynchronous execution model. Also, since we define streams as FIFO queues to submit computation tasks in our system, command buffers act as the role of execution boundaries and barriers. We can safely reorder commands within a command buffer as long as the dependencies between commands are preserved, achieving higher hardware utilization.

By exploiting the closed-world knowledge of the resource usage in kernels and commands, we design a algorithm to automatically perform the scheduling, without the distracting manual dependency marking or barrier placement as commonly requested by modern graphics APIs. The algorithm is efficient and capable to process thousands of commands in a millisecond. For example, on M1 Max, it schedules 1045 commands submitted at once inside a single command buffer in 0.86ms for the wavefront path integrator.

Moreover, the tracking is performed at the sub-resource level, so that false-positive dependencies, such as writes to non-overlapping ranges within the same buffer, are effectively culled, thus offering a broader space for the discovery of possibly higher concurrency. An illustration is given in Fig. 3.

6 BACKEND IMPLEMENTATION DETAILS

While we extract the common concepts for computing devices and encapsulate them in a unified runtime layer, there is still a lot of special care and effort to pay when implementing the abstract interfaces atop different low-level backend APIs.

Currently, we have implemented 5 backends based on CUDA, Metal, DirectX, ISPC (with vectorization on CPUs), and LLVM (scalar CPU), respectively. In this section, we will explain several technical details of the backend implementations.

6.1 Native API Adaptation

To correctly and effectively implement the runtime interfaces, we have to wrap the native APIs, manage the resources, ensure data layouts, respect the execution order, etc. For example, in the CUDA backend, ray tracing shaders are taken over and spawned by OptiX rather than native CUDA. Therefore, we implement a mechanism to automatically detect ray tracing calls in the shader, which, if found, triggers the different code path for code generation and compilation, pipeline and shader binding table construction, parameter buffer preparation, and kernel launching. For the DirectX backend, we need to wrap the pipeline-based low-level API into the more semantic, stateless task-based interfaces and implement resource trackers and barriers, while simultaneously maintaining an decent performance.

Listing 15 gives the translation process of the buffer-to-texture copying command in LuisaRender to the native CUDA driver API. For more complicated commands (such as acceleration structure construction and kernel launching) on lower-level APIs (e.g., DirectX), the elimination of boilerplate code appears even more significant.

6.2 Code Generation

For the CUDA, Metal, DirectX, and ISPC backends, code generation is performed in a source-to-source fashion, i.e., from the DSL to the platform-specific native shading languages. For LLVM, we leverage its built-in `IRBuilder` to construct the in-memory IR modules.

The usage of ASTs helps preserve kernel structures and semantics, simplifying the translation process. Still, we have to work around

```
// with LuisaRender
stream << texture.copy_from(buffer);

// with native CUDA
CUDA_MEMCPY3D copy{};
copy.srcMemoryType = CU_MEMORYTYPE_DEVICE;
copy.srcDevice = /* buffer address */;
copy.srcPitch = /* texture pitch (in bytes) */;
copy.srcHeight = /* texture height (in texels) */;
copy.dstMemoryType = CU_MEMORYTYPE_ARRAY;
copy.dstArray = /* texture handle */;
copy.WidthInBytes = /* texture pitch (in bytes) */;
copy.Height = /* texture height (in texels) */;
copy.Depth = /* texture depth (in texels) */;
cuMemcpy3DAsync(&copy, /* stream handle */);
```

Listing 15. Translation of the buffer-to-texture copying command in LuisaRender to the native CUDA driver API.

defects and complement missing functionalities in the backend shading languages and compilers. For example, the ISPC compiler does not allow visiting structure members of rvalues, so we detect and have special handling for such cases by transforming them into built-in function calls. ISPC is also sensitive to variables defined in outer scopes but used in branches because it has to generate SIMD masks for their `loads` and `stores`. Therefore, we analyze and tweak the definition site of each local variable, narrowing its lifetime scopes and decreasing the number of necessary masks, which brings up to 100× faster compilation and 5× runtime performance boost (e.g., on M1 Max with the mega-kernel path integrator, this optimization reduces the compilation time from 2152.3s to 23.9s for the *Coffee* scene and the 1024spp rendering time decreases from 2627.2s to 493.6s at the resolution of 1200×1800).

Data layouts are another important aspect to pay attention to. We allow users to specify the alignments of structures, which is easily realizable in Metal and CUDA with the built-in support for the `alignas` specifier. However, such language constructs are missing in HLSL and ISPC, so we have to manually pad the structures to achieve a conforming layout. Moreover, the intricate constant buffer packing rules in DirectX requires correct transformation of the input host data. We bypass these rules by using argument buffers.

6.3 Shader Compilation

The generated shader sources are JIT compiled, typically via in-memory compilation interfaces (e.g., LLVM MCJIT, NVRTC, and the DirectX Shader Compiler) to elide disk I/O. We ensure a thread-safe design to allow accelerated parallel compilation with multithreading. Also, by employing memory and disk caches, redundant code generation and compilation for identical kernels are eliminated, further reducing the runtime overhead.

7 APPLICATIONS AND EXTENSIONS

7.1 Physically-Based Offline Renderer

To better understand and demonstrate the practicality of LuisaRender, we build a high-performance cross-platform physically-based offline renderer atop, utilizing almost all the available features, constructs, and patterns provided by the framework.

Architecture Design. The high-level interfaces of the renderer are expressed as a node system, where the scenes are organized as graphs with component nodes, such as cameras, shapes, and materials, each tracking reference to child nodes as well as necessary parameters and resource descriptors (e.g., paths to textures). To maximize the extensibility of the renderer, all the components are implemented as decoupled, dynamically loaded plug-ins, so that users can freely add new functionalities to the renderer without modifying and rebuilding the entire application.

After loading the scene, a build process instantiates the scene graph into a data-oriented pipeline on a specified backend device. With the embedded DSL enabling the powerful multiple-stage programming capability (discussed in Sec. 4.2), the scene components, each responsible for a modular functionality in the renderer, are actually playing the roles of AST builders, expanding the corresponding logic. The system fuses the nodes into flattened rendering kernels, while automatically tracking resources in use and scheduling an optimized execution order.

In other words, we provide a flexible and easy-to-use interface and clean implementation comparable to research- and education-oriented CPU renderers that heavily exploit dynamism and polymorphism, while achieving much better performance across platforms. Sec. 8 gives the comparison with several popular open-source research and production renderers.

Differentiable Rendering. Beyond the conventional forward rendering workloads, inverse rendering, typically via physically based differentiable rendering techniques, is arousing more and more attention and interest both in academia and industry nowadays. Therefore, we also equip our renderer with two differential light transport algorithms, namely *Radiative Backpropagation* [Nimier-David et al. 2020] and *Path Replay Backpropagation* [Vicini et al. 2021], both implemented as integrator plug-ins. We showcase an inverse rendering example in Sec. 8.2.2.

7.2 Python Frontend

The layered architecture of our system well decouples different levels of abstraction and thus enables extending and replacing specific layers while reusing the others. We provide a Python frontend for fast prototyping, which binds to the same runtime, AST, and backend layers as the C++ frontend but replaces the embedded DSL and resource wrappers with a re-implementation in pure Python.

Being a dynamic language, Python allows us to inspect the user source code and parse it into ASTs. Therefore, different from the tracing strategy in C++, we directly obtain and traverse the ASTs of Python functions and classes, and translate them into the internal AST representations in LuisaRender through the exported `FunctionBuilder`. Resource wrappers, as well as other high-level logic and user-side interfaces, are also implemented within Python in a "Pythonic" way, whose functionalities are eventually converted to lower-level API calls forwarded to the core system in C++.

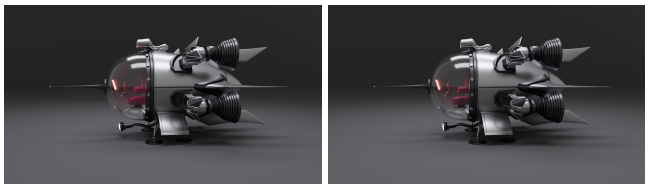
The Python frontend achieves a similar performance to the native C++ implementation on dense, compute-heavy workloads, outperforming existing frameworks such as Taichi [Hu et al. 2019]. Meanwhile, it provides richer constructs necessary in rendering than existing frameworks, such as textures and acceleration structures,

```python
import luisa
from luisa.mathtypes import *
from cv2 import imwrite

n = 2048
luisa.init()
image = luisa.Texture2D.zeros((2 * n, n), 4, float)

@luisa.func # makes LuisaRender handle the function
def draw(max_iter):
    p = dispatch_id().xy
    z, c = float2(0), 2 * p / n - float2(2, 1)
    for itr in range(max_iter):
        z = float2(z.x**2 - z.y**2, 2 * z.x * z.y) + c
        if length(z) > 20:
            break
    image.write(p, float4(float3(1 - itr/max_iter), 1))

draw(50, dispatch_size=(2*n, n)) # parallelize
imwrite("mandelbrot.exr", image.numpy())
```

Listing 16. Example user code with the Python frontend.

Fig. 4. Some of the test scenes: *Glass of Water*, *Spaceship*, *Coffee*, *Staircase*, and *Dining Room*. See also Fig. 1 for two other test scenes, *Classroom* and *Dining Room*. All scenes are from *Rendering Resources* [Bitterli 2016].

upon which rapid development of graphics applications, such as simple path tracers and shader toys, are made easy and convenient without loss of performance, as demonstrated in Sec. 8. An example program with the Python frontend is shown in Listing 16.



(a) Ours (b) Mitsuba 3

Fig. 5. Rendering comparison between (a) our renderer and (b) Mitsuba 3. Both applications use the spectral mode and render the *Spaceship* scene with 16 bounces and 16384spp at 1080p. The renderings are visually identical with minor numeric difference (1-SSIM: 0.01148).

8 EXPERIMENTS

In this section, we conduct several experiments to measure and analyze the performance of LuisaRender, as well as comparing it with other state-of-the-art graphics systems.

8.1 Test Scenes and Platforms

We adapt 7 test scenes from the *Rendering Resources* website [Bitterli 2016]: *Coffee*, *Glass of Water*, *Classroom*, *Staircase*, *Spaceship*, *Living Room*, and *Dining Room*. Fig. 4 shows a snapshot of the test scenes.

The experiments are conducted on multiple platforms. Tested CPUs include Intel i9-9900K (8-core 16-thread) and Apple M1 Max (with 8 performance cores and 2 efficiency cores). Tested GPUs include NVIDIA RTX-2080Ti (11GB), RTX-3080Ti (12GB), and Apple M1 Max (32-core). The PCs feature 32GB memory and the Mac has 64GB unified memory shared across the CPU and GPU.

8.2 Comparisons

8.2.1 Comparison with other renderers. To showcase the performance of our framework, we compare the rendering time of the test scenes using our renderer and existing open-source renderers including PBRT-v4 [Pharr et al. 2016], Mitsuba 3 [Jakob et al. 2022b], Falcor [Kallweit et al. 2017] and Ignis [Ignis 2022]. Note that Ignis is a follow-up renderer based on the Rodent framework [Pérard-Gayot et al. 2019], with shorter compilation time, better performance and more supported scenes. Besides the research renderers, we also compare our renderer with Cycles, which is a GPU-enabled production renderer from the Blender [Blender Online Community 2022] project. We run all the tests on Windows 11, except for Ignis on Linux (Manjaro distribution with kernel version 5.15) which we fail to make work on Windows.

All scenes are rendered at 1080p, except *Coffee* at 1200×1080. We tune the scenes and renderer options to ensure a fair comparison. Fig. 5 compares the *Spaceship* renderings of our renderer and Mitsuba 3. The visual appearances are almost identical, despite the minor numeric difference due to the implementation discrepancies between renderers, e.g., in the material and shading systems.

The results of 8-bounce renderings are displayed in Fig. 6, configurations including all combinations of mega-kernel/wavefront (solid/striped bars), RGB/spectral, and CPU/GPU. Note that our renderer supports all the configurations, while other renderers may

not support certain ones. We also experiment with 16-bounce renderings and the trends appear similar. We refer the audience to the supplemental material for more details.

On GPUs, we run our renderer on two backends, CUDA and DirectX, and our renderer significantly outperforming PBRT-v4, Mitsuba 3, and Ignis in all scene configurations on both backends. Falcor runs faster than other renderers but is still slower than our DirectX mega-kernel counterpart. On CPUs, our renderer is much faster than PBRT-v4 but slower than Mitsuba 3 in most cases. We conclude that the array model used in Mitsuba 3 fits well for vectorized computation on CPUs where the SIMD width is relatively small (typically 4 or 8).

The runtime comparisons show that our framework is able to generate highly optimized programs at the scale of intricate offline rendering tasks. We also analyze the results on different platforms and backends, and between the mega-kernel version and the wavefront version. Please refer to Sec. 8.3.

8.2.2 Comparison on differentiable rendering. *Radiative Backpropagation* [Nimier-David et al. 2020] (RB) is a differential light transport algorithm for material optimization. *Path Replay Backpropagation* [Vicini et al. 2021] (PRB) further improves RB to compute unbiased gradients of material parameters. We implement both algorithms (without support for volumes and perfect specular surfaces) in our renderer as integrator plug-ins. In Fig. 7, we compare the performance of our PRB implementation with the one in Mitsuba 3. Our implementation is about 4× faster, both in the forward and backward rendering processes.

8.2.3 Comparison with Taichi. Taichi [Hu et al. 2019] is a graphics-oriented DSL embedded in Python. We port two example applications from Taichi to our framework: *MPM 3D*, which simulates fluids with the material point method in dense grids; and *SDF Renderer*, which path-traces a signed distance field. As shown in Fig. 8, we compare the performance of the examples on Taichi, our C++ DSL, our Python DSL, and native CUDA. All Taichi's and our programs are tested on the CUDA backend on RTX-3080Ti. Our framework consistently outperforms Taichi in supporting these applications of dense computation, achieving a closer frame time to the native CUDA implementations. The results indicate that our system is able to generate CUDA code with a comparable quality to the hand-tuned ones. Also, we notice that when the dispatch sizes are small (e.g., 64^3 grids in *MPM 3D*), the overhead of our Python frontend is larger, but still smaller than that of Taichi.

8.3 Analysis

8.3.1 Backends and devices. We implement multiple backends in LuisaRender to support various hardware/OS platforms. Fig. 9 compares the rendering time on different backends and platforms. On PCs with NVIDIA graphics cards where CUDA and DirectX are available, we found that the DirectX backend runs consistently faster than CUDA. This is probably because DirectX is a lower-level API featuring *inline ray tracing*, thus having less overhead. On Macs, the Metal backend is available for GPUs. However, without dedicated ray-tracing hardware units, the M1 Max GPU is much slower than the RTX ones, which also reflects the necessity of the built-in support for acceleration structures in our framework.

8.3.2 Mega-kernel vs. wavefront. *Wavefront path tracing* [Laine et al. 2013] separates the computation stages (e.g. ray generation, intersection, next-event estimation, and shading) into multiple kernels, as opposed to *mega-kernels*, where a monolithic kernel handles a full path tracing loop per thread. The major advantage of wavefront path tracing is the improved thread coherence, since threads are at the same computation stage, but at the cost of increased global memory I/O and kernel launches. As a result (Fig. 9), we see wavefront path tracing achieving twice the performance compared to mega-kernels on the M1 Max GPU, which is more sensitive to thread divergence. However, small or negative improvements are observed on the newer hardware RTX-3080Ti, possibly due to the 2nd-generation RT-cores with optimized hardware scheduling and the ability of concurrent shading and tracing.

Notably, the wavefront model appears more profitable in the spectral rendering cases, where the computation requires more registers and is more likely to cause register spilling in mega-kernels. By contrast, the wavefront implementation splits the rendering process into smaller kernels and thus alleviates the register pressure.

8.3.3 De-virtualization. In this experiment we showcase the performance benefit of de-virtualization as mentioned in Sec. 4.4, one of the most important optimizations enabled by our system. De-virtualizing polymorphic function calls drastically reduces branches and register pressure caused by dynamic dispatching. As shown in Fig. 10, this optimization reduces rendering time by up to 39%.

8.3.4 Command scheduling. As discussed in Sec. 5.3, our framework can automatically schedule commands within a command buffer based on the resource usage information in kernels and commands. The algorithm re-groups non-dependent commands into sub-lists and submits them to the device at once, which potentially reduces resource barriers and state changes and improves hardware utilization. In Fig. 11, we compare the rendering time with and without (i.e., fencing each command into individual lists) automatic command reordering. The results show that command scheduling reduces rendering time by up to 19%.

8.3.5 Preparation time. Our renderer has to go through a few preparation steps before rendering a scene, which mainly include

(1) Scene loading, where scene files are parsed and required plug-ins (e.g., shapes, cameras, and integrators) are loaded;
(2) Pipeline creation that creates devices resources (e.g., buffers, textures, and meshes) and uploads data to them;
(3) Shader compilation that generates backend-specific shader code from kernel ASTs and compiles it to executable shaders that are also optionally cached for future use; and on
(4) Shader cache hit, i.e., when compilation cache is found for a shader, we can skip the compilation step.

We measure and list the detailed time of each step in Table 1 for the *Coffee* scene. When cache is not available, shader compilation takes up most of the time. Compilation on CUDA, DirectX, and LLVM is decently fast. The ISPC backend is much slower at compiling shaders but the time is significantly reduced with cache. Aside

Fig. 6. Time (in seconds) to render 7 different scenes with different renderers in all combined integrator and spectrum configurations. Solid-colored and striped bars refer to the *mega-kernel* and *wavefront* modes, respectively. All scenes are rendered on an i9-9900K (CPU) or RTX-3080Ti (GPU) with 8 bounces and 1024spp. The *Coffee* scene is at 1200 × 1800 and other scenes are at 1080p. Results with 16 bounces are included in the supplemental material.

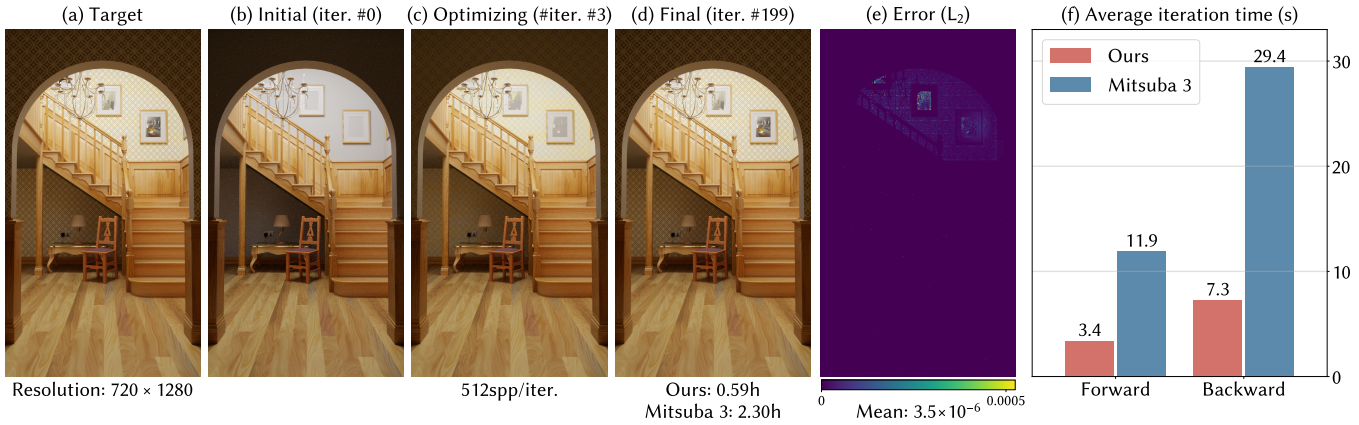


Fig. 7. Average iteration time (in seconds, lower is better) to optimize the albedo textures of the wallpaper and the decorative paintings on the wall on CUDA using an L_2 objective function. The scene is rendered in RGB at 720 × 1280 pixels with 512spp per forward/backward pass. Both renderers use the CUDA backends on RTX-2080Ti with the mega-kernel mode. The L_2 error between the final image (d) after 199 iterations and the target image (a) is shown in (e).

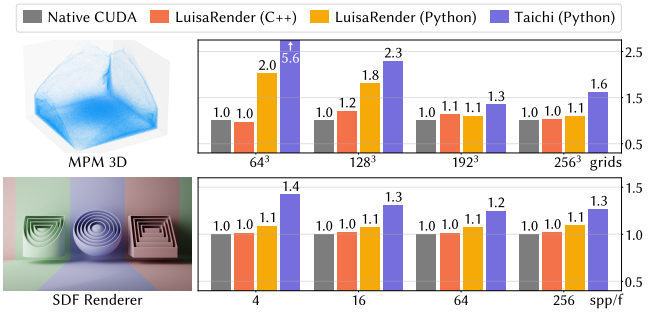


Fig. 8. Frame time of *MPM 3D* and *SDF Renderer*, relative to the native CUDA implementations. Lower is better. *MPM 3D* simulates 256 iterations with 25 steps per iteration. *SDF Renderer* runs at 1280×720 pixels. Both programs run on RTX-3080Ti with display disabled.

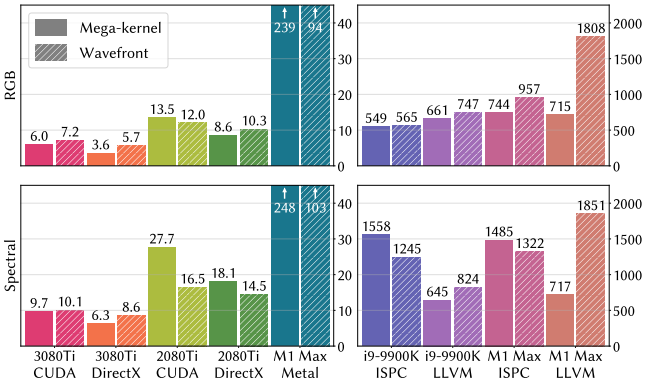


Fig. 9. Rendering time (in seconds) of *Staircase* on different platforms and backends. The scene is rendered with 8 bounces and 1024spp at 1080 × 1920.

these backends, Metal has a system-controlled compilation cache and thus is not included in the table.

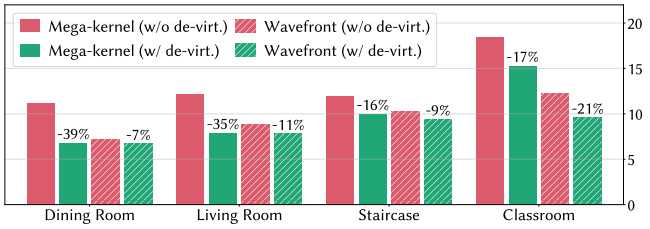


Fig. 10. Rendering time (in seconds) on RTX-3080Ti with or without de-virtualization. All scenes are spectrally rendered on CUDA with 8 bounces and 1024spp at 1080p.

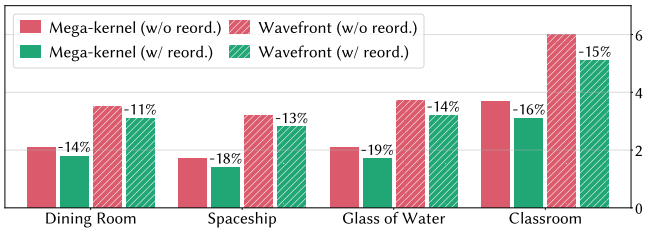


Fig. 11. Rendering time (in seconds) on RTX-3080Ti with or without automatic command scheduling. All scenes are rendered in RGB on DirectX with 8 bounces and 1024spp at 1080p.

8.3.6 Scalability. To test the scalability of our renderer, we stress it with varying numbers of objects, light sources, and unique material instances, respectively. In each test, we fix other parameters except the corresponding testees, and measure the compilation and rendering time. Specifically, in the light test, there are 400 objects assigned 20 materials, and we increase the number of lights from 20 to 1000 and assign 15 emissive surface profiles to them. In the material test, there are 15 lights each with a unique profile and 400 objects to which we increasingly assign 5 to 200 unique material instances. In the object test, we fix the number of lights to 15, each with a

Table 1. Breakdown of preprocessing time (in seconds) of the *Coffee* scene, using the mega-kernel path tracer in RGB. The timings are measured on a Windows PC with i9-9900K and RTX-2080Ti. The AST construction time is negligible (typically under or about a millisecond), thus not reported.

Backend	Scene Loading	Pipeline Creation	Shader Compilation	Shader Cache Hit
CUDA	0.26	0.08	2.57 (NVRTC)	0.014
DirectX	0.26	0.17	0.54 (DXC)	—
ISPC	0.29	0.12	35.11 (ISPC)	0.09
LLVM	0.28	0.07	1.04 (MCJIT)	0.26

unique profile, and the number of material instances to 20. The objects count from 100 to 1000, assigned the 20 material instances. In all cases, a sphere mesh is used for objects. We arrange them in a $10 \times 10 \times 10$ array by filling the x, z, and y axes in order.

The results are shown in Fig. 12, from which we have the following interesting observations:

- The wavefront mode is more efficient than mega-kernel for relatively large numbers of lights, materials, and objects, due to the reduced thread divergence and register pressure;
- The compilation and rendering time is insensitive to the numbers of lights with fixed numbers of objects, material instances, and emissive surface profiles;
- The compilation time increases linearly with the number of material instances that add dynamic branches to the pipeline, while the rendering time grows steadily; and
- The rendering time has a relatively obvious growth with the object counts, which possibly results from the deeper BVHs and the longer paths before the rays exit the scene due to the increased inter-reflections between objects.

9 CONCLUSION AND FUTURE WORK

In this paper, we present LuisaRender, a high-performance rendering framework for modern stream architectures, with the objective to seek balance between unification, programmability, and performance. The framework features an expressive DSL embedded in modern C++ to ease kernel authoring, a unified runtime with user-friendly resource wrappers to hide device-specific and low-level details, and multiple backends on different devices and graphics APIs. We also implement a cross-platform Monte Carlo renderer on the framework. As demonstrated in the experiments, it achieves significantly higher performance on modern graphics hardware than state-of-the-art open-source renderers.

We plan to further improve the usability of our framework on different platforms and use scenarios by supporting more backends in the future. Example can be a Vulkan backend for even better controllability and portability across multiple GPU platforms, or a proxy backend for remote and/or distributed computation. As gradient-based research prevailed in recent years, functionalities of automatic differentiation may also be added to our framework, to support the research and development of differentiable rendering systems and deep learning systems. Other optimizations and improvements for domain-specific usages are also possible, e.g.,

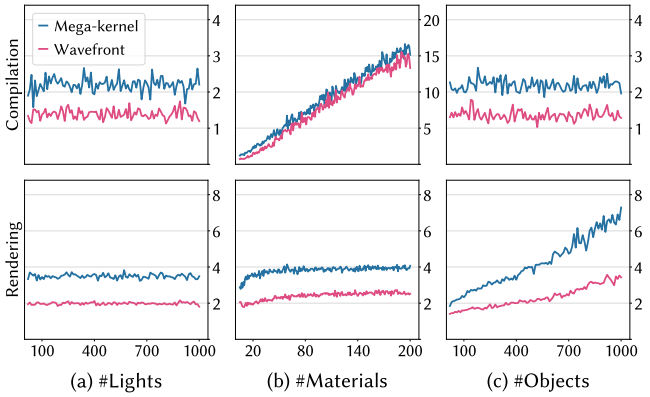


Fig. 12. Compilation (top row, in seconds) and rendering time (bottom row, in seconds) of the scalability tests. The timings are measured on RTX-3080Ti with the DirectX backend. (a) Increasing light instances (20–1000), with fixed numbers of 400 objects, 20 material instances, and 15 emissive surface profiles. (b) Increasing unique material instances (5–200) assigned to a fixed number of 400 objects, which are lit by 15 light instances each assigned a unique emissive profile. (c) Increasing objects (20–1000), with fixed numbers of 20 material instances and 15 lights each assigned a unique profile. We render all cases in RGB with 8 bounces and 1024spp at 1024×1024.

support for sparse data structures in physical simulation. In addition, the Python frontend may be further exploited, to supply an easy-to-use high-performance computing framework to a broader audience other than rendering, potentially packing interoperability with machine learning packages in Python (e.g., PyTorch [Paszke et al. 2019]).

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their insightful comments and valuable suggestions. This work is supported by the National Natural Science Foundation of China (Project Number: 61932003). Yan Gu is supported by NSF CCF #2103483.

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. http://tensorflow.org/ Software available from tensorflow.org.

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 109 (oct 2021), 28 pages. https://doi.org/10.1145/3485486

Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. 2017. Aether: An Embedded Domain Specific Sampling Language for Monte Carlo Rendering. *ACM Trans. Graph.* 36, 4, Article 99 (jul 2017), 16 pages. https://doi.org/10.1145/3072959.3073704

Apple. 2021. Metal. https://developer.apple.com/metal/

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

Blender Online Community. 2022. *Blender - A 3D Modelling and Rendering Package*. Blender Foundation, Stichting Blender Foundation, Amsterdam. http://www.blender.org

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. *SIGPLAN Not.* 48, 6 (jun 2013), 105–116. https://doi.org/10.1145/2499370.2462166

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, Article 171 (oct 2017), 27 pages. https://doi.org/10.1145/3132188

Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *J. Mach. Learn. Res.* 17, 1 (jan 2016), 2909–2913.

Epic Games. 2019. *Unreal Engine*. https://www.unrealengine.com

Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Trans. Graph.* 37, 3, Article 31 (aug 2018), 18 pages. https://doi.org/10.1145/3182161

Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. https://mlsys.org/Conferences/doc/2018/146.pdf

Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: Language Mechanisms for Extensible Real-Time Shading Systems. *ACM Trans. Graph.* 37, 4, Article 141 (jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201380

Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. 2017. Shader Components: Modular and High Performance Shader Development. *ACM Trans. Graph.* 36, 4, Article 100 (jul 2017), 11 pages. https://doi.org/10.1145/3072959.3073648

Felix Heide, Steven Diamond, Matthias Nießner, Jonathan Ragan-Kelley, Wolfgang Heidrich, and Gordon Wetzstein. 2016. ProxImaL: Efficient Image Optimization Using Proximal Algorithms. *ACM Trans. Graph.* 35, 4, Article 84 (jul 2016), 15 pages. https://doi.org/10.1145/2897824.2925875

Shi-Min Hu, Dun Liang, Guo-Ye Yang, Guo-Wei Yang, and Wen-Yang Zhou. 2020b. Jittor: a novel deep learning framework with meta-operators and unified graph execution. *Science China Information Sciences* 63, 222103 (2020), 1–21.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020a. DiffTaichi: Differentiable Programming for Physical Simulation. In *Proceedings of ICLR 2020*.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Trans. Graph.* 40, 4, Article 182 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459671

Ignis. 2022. *Ignis*. https://github.com/PearCoding/Ignis

Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. 2022b. *Mitsuba 3 renderer*. https://mitsuba-renderer.org.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022a. DR.JIT: A Just-in-Time Compiler for Differentiable Rendering. *ACM Trans. Graph.* 41, 4, Article 124 (jul 2022), 19 pages. https://doi.org/10.1145/3528223.3530099

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. 2017. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor https://github.com/NVIDIAGameWorks/Falcor.

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference* (Anaheim, California) *(HPG '13)*. Association for Computing Machinery, New York, NY, USA, 137–143. https://doi.org/10.1145/2492045.2492060

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3105762.3105768

Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (oct 2018), 30 pages. https://doi.org/10.1145/3276489

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201383

William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph.* 22, 3 (jul 2003), 896–907. https://doi.org/10.1145/882262.882362

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany) *(HWWS '02)*. Eurographics Association, Goslar, DEU, 57–68.

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. 2020. Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering. *ACM Trans. Graph.* 39, 4, Article 146 (jul 2020), 15 pages. https://doi.org/10.1145/3386569.3392406

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Trans. Graph.* 38, 6, Article 203 (nov 2019), 17 pages. https://doi.org/10.1145/3355089.3356498

NVIDIA. 2022. NVIDIA Warp. https://developer.nvidia.com/warp-python

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 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: Generating Renderers without Writing a Generator. *ACM Trans. Graph.* 38, 4, Article 40 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322955

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. https://github.com/mmp/pbrt-v4

Matt Pharr and William R. Mark. 2012. Ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13. https://doi.org/10.1109/InPar.2012.6339601

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.* 31, 4, Article 32 (jul 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Kerry A. Seitz, Tim Foley, Serban D. Porumbescu, and John D. Owens. 2019. Staged Metaprogramming for Shader System Development. *ACM Trans. Graph.* 38, 6, Article 202 (nov 2019), 15 pages. https://doi.org/10.1145/3355089.3356554

Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

Delio Vicini, Sébastien Speierer, and Wenzel Jakob. 2021. Path Replay Backpropagation: Differentiating Light Paths Using Constant Memory and Linear Time. *ACM Trans. Graph.* 40, 4, Article 108 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459804

Fahad Zafar, Marc Olano, and Aaron Curtis. 2010. GPU Random Numbers via the Tiny Encryption Algorithm *(HPG '10)*. Eurographics Association, Goslar, DEU, 133–141.