GPU Coroutines for Flexible Splitting and Scheduling of Rendering Tasks

SHAOKUN ZHENG, BNRist, Department of CS&T, Tsinghua University, China
XIN CHEN, BNRist, Department of CS&T, Tsinghua University, China
ZHONG SHI, BNRist, Department of CS&T, Tsinghua University, China
LING-QI YAN, University of California, Santa Barbara, United States of America
KUN XU*, BNRist, Department of CS&T, Tsinghua University, China

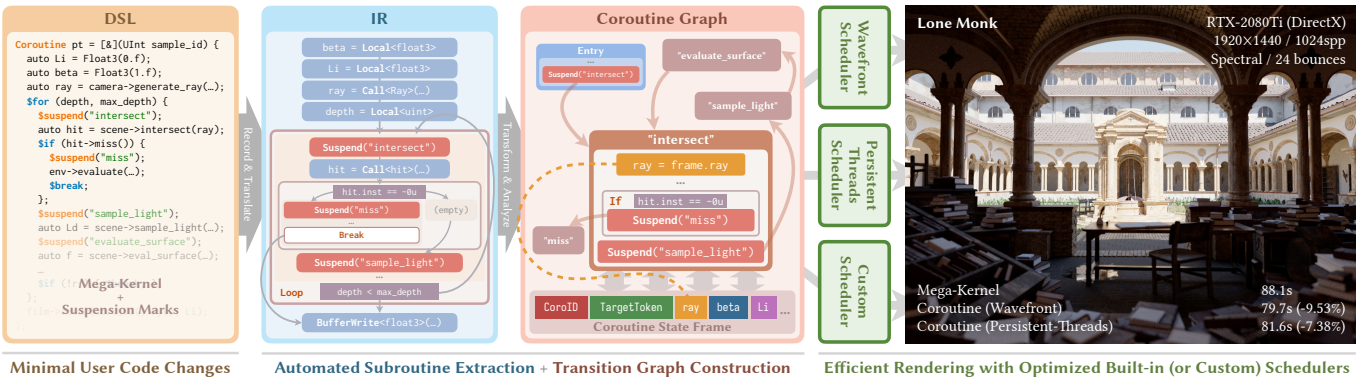


Fig. 1. We incorporate the coroutine concept into GPU kernel programming and reify it with language constructs and multiple built-in schedulers, facilitating the splitting and scheduling of complex rendering tasks. Our implementation extends Luisa's domain-specific language (DSL) and runtime constructs. Users can write the rendering algorithm in a mega-kernel fashion with suspension marks to define a coroutine function, which is dynamically recorded and translated into an intermediate representation (IR). Suspension is allowed in arbitrarily nested control flow, ensuring flexibility with ease of programming. We then perform a set of compiler transformation and analysis passes to automatically extract the continuation subroutines of each suspension point and materialize the state frame, both encoded into a graph representation. Schedulers, either built-in or customized, can be freely selected to execute the subroutines and manage the state frames. Experiments on intricate rendering tasks, including path tracing, demonstrate the potential of our GPU coroutines.

We introduce *coroutines* into GPU kernel programming, providing an automated solution for flexible splitting and scheduling of rendering tasks. This approach addresses a prevalent challenge in harnessing the power of modern GPUs for complex, imbalanced graphics workloads like path tracing. Usually, to accommodate the SIMT execution model and latency-hiding architecture, developers have to decompose a monolithic mega-kernel into smaller sub-tasks for improved thread coherence and reduced register pressure. However, involving the handling of intricate nested control flows and numerous interdependent program states, this process can be exceedingly tedious and error-prone when performed manually.

Coroutines, a building block for asynchronous programming in many high-level CPU languages, exhibit untapped potential for restructuring GPU kernels due to their versatility in control representation. By extending Luisa [Zheng et al. 2022], we implement an asymmetric, stackless coroutine model with programming language support and multiple built-in schedulers for modern GPUs. To showcase the effectiveness of our model and implementation, we examine them in different application scenarios, including path tracing, SDF rendering, and incorporation with custom passes.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Parallel programming languages**; • **Software and its engineering** → **Domain specific languages**; **Compilers**; **Coroutines**.

Additional Key Words and Phrases: GPU, Coroutine, Rendering, Program Transformation, Asynchronous Programming, Task Scheduling

ACM Reference Format:
Shaokun Zheng, Xin Chen, Zhong Shi, Ling-Qi Yan, and Kun Xu. 2024. GPU Coroutines for Flexible Splitting and Scheduling of Rendering Tasks. *ACM Trans. Graph.* 43, 6, Article 281 (December 2024), 24 pages. https://doi.org/10.1145/3687766

1 INTRODUCTION

Modern GPUs have evolved into powerful general-purpose parallel computing devices, capable of handling increasingly complex rendering tasks. However, although feasible and convenient, implementing a heavy workload such as path tracing in a monolithic mega-kernel is usually not the optimal solution. Low hardware utilization may occur, due to the overwhelming thread divergence in both control flow and memory access.

*Kun Xu is the corresponding author.

Authors' addresses: Shaokun Zheng, BNRist, Department of CS&T, Tsinghua University, Beijing, China, zsk20@mails.tsinghua.edu.cn; Xin Chen, BNRist, Department of CS&T, Tsinghua University, Beijing, China, xin-chen22@mails.tsinghua.edu.cn; Zhong Shi, BNRist, Department of CS&T, Tsinghua University, Beijing, China, shizhong24@mails.tsinghua.edu.cn; Ling-Qi Yan, University of California, Santa Barbara, Santa Barbara, United States of America, lingqi@cs.ucsb.edu; Kun Xu, BNRist, Department of CS&T, Tsinghua University, Beijing, China, xukun@tsinghua.edu.cn.

0730-0301/2024/12-ART281
https://doi.org/10.1145/3687766

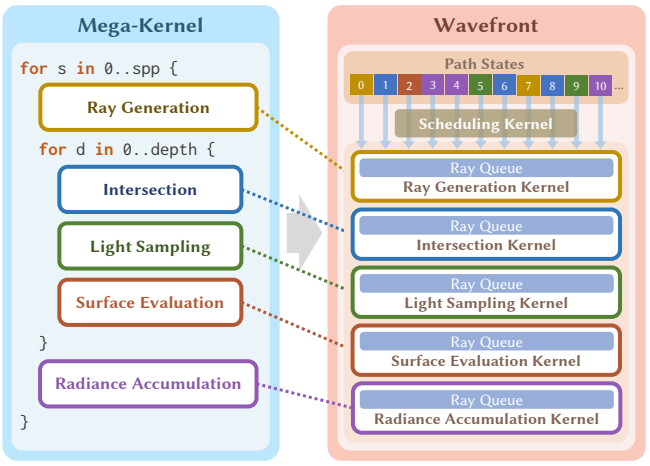


Fig. 2. Wavefront path tracing decomposes the path tracing logic into smaller sub-kernels, each handling a rendering stage. A scheduling kernel is employed to sort the path states into queues for the sub-kernels. The batched processing of similar paths improves thread coherence on GPUs.

For such reasons, graphics developers often have to refactor a rendering task into multiple smaller GPU kernels and carefully plan their execution for improved performance. A well-known example is *wavefront path tracing* [Laine et al. 2013], which decomposes the mega-kernel path tracing process into multiple sub-kernels. At each iteration, a scheduler kernel sorts the paths into per-kernel queues for coherent batched tracing and shading (Fig. 2).

However, manually splitting and scheduling mega-kernels is rather complicated and error-prone. Restructuring of intricate nested control flows, collection of numerous states scattered across modules, management of various runtime resources, etc., are each hard enough to handle, not to mention the deep coupling in between.

Such case-by-case tedious labor prompts us to seek a unified, automated solution, which would necessarily involve three parts:

(1) Code transformation that automatically decomposes the program into subroutines with user-specified marks;
(2) Data-flow analysis that determines the live states across splitting points and their usage in each subroutine; and
(3) Built-in schedulers that efficiently execute the resulting subroutines and manage the live states;

so that no longer would and should developers be distracted by manual refactoring of sophisticated rendering tasks. Instead, they could concentrate on the expression of the algorithm in a mega-kernel manner. The underlying framework would take over the most annoying code transformation work and provide various scheduling schemes for them to try out.

We observe that these demands are closely linked to the concept of *coroutines*, an asynchronous programming construct widespread in high-level CPU languages, typically implemented as suspendable functions. The suspension operation captures the current program state and remaining instructions into a closure, known as the *continuation*, and then transfers control back to the caller. When the continuation is later invoked, it resumes execution from the suspension point, as if the program had never stopped. With both logic and data encapsulated in the closure, the resumption can be strategically deferred, thus providing flexibility in scheduling.

Incorporating this construct into GPU programming, suspension points can act as execution barriers that divide a monolithic kernel into smaller sub-stages. The resumption of these sub-stages can then be rearranged by suitable schedulers at opportune times, thereby enhancing performance.

However, despite their established presence in many CPU languages, adapting coroutines for GPU programming is non-trivial due to the distinct architectures:

- Opaque hardware-managed thread context. Unlike CPUs, which usually allow direct manipulation of the call stack via their instruction sets, GPUs manage thread contexts at the hardware level, making them inaccessible to user programs. A purely user-space implementation is required.
- Critical per-thread resource efficiency. The massive parallelism of GPUs makes it crucial to manage coroutine frames efficiently. Reducing per-thread register usage and memory bandwidth is essential to maximize hardware utilization.
- Multi-level asynchronicity. Coroutines' deferred resumption adds another level of asynchronicity to the already asynchronous GPU execution, necessitating specialized schedulers for effective latency hiding and resource management.

Therefore, a suitable GPU coroutine model must be carefully chosen, accompanied by pragmatic, hardware-oriented optimizations in both the language and runtime implementation.

We opt for an asymmetric, stackless coroutine model. We base the coroutine language features and runtime support on Luisa, to reuse its constructs while avoiding platform-specific details. The implementation not only supports traditional asynchronous programming patterns, such as generators and `await`, but is also tailored for compute-intensive tasks on massively parallel GPU hardware with multiple built-in schedulers. As shown in Fig. 3, with the proposed language and runtime features, rewriting a mega-kernel can be accomplished in two simple, decoupled steps: marking suspension points and handing it over to a chosen scheduler, without any kernel logic modification.

To demonstrate the effectiveness of the GPU coroutine concept and to examine the practicality of our implementation, we conducted experiments across multiple applications and settings. We transformed path tracing and SDF rendering applications, originally written as mega-kernels, into versions utilizing coroutines and measured the performance gains with the built-in schedulers. Additionally, we showed that GPU coroutines can serve as a general control-flow restructuring tool, simplifying the incorporation of external computation passes during the execution of a mega-kernel.

In summary, our contributions include

(1) The introduction of the coroutine concept into GPU programming, observing and highlighting its deep connections to kernel splitting and scheduling techniques;
(2) A systematic design of the coroutine model that adapts well to modern GPUs, along with a practical implementation of language constructs and runtime support; and

(3) An exploratory study of using coroutines in massively parallel computing, demonstrated through several representative rendering applications.

2 RELATED WORK AND BACKGROUND

2.1 GPGPU Programming

Programmability is an everlasting topic in the evolution of GPUs. Even before the birth of programmable graphics hardware and APIs, pioneering endeavors have modeled early GPUs as SIMD computers and compiled procedural shaders to combinations of multiple fixed-function rendering passes [Lastra et al. 1995; Olano and Lastra 1998; Peercy et al. 2000]. Entering the programmable era, various efforts were made to bring high-level language features and interfaces to rasterization shader programming [Mark et al. 2003; McCool et al. 2002; Proudfoot et al. 2001]; or to explore the potential patterns for generic computation [Buck et al. 2004; Hou et al. 2008].

Today, graphics processors are not just specialized rasterizers but powerful general-purpose computing hardware, thus often referred to as general-purpose GPUs (GPGPUs). They leverage the *single instruction, multiple threads* (SIMT) execution model to achieve massive parallelism and latency hiding, where a single program (kernel) is dispatched to many lightweight GPU threads to process multiple data streams simultaneously. In companion, modern shading languages (such as HLSL, GLSL, CUDA C++, and Metal Shading Language) are high-level C/C++-style imperative languages with tailored syntax and specialized standard libraries for this parallel SIMT programming model and GPU-specific functionalities.

As co-processors, GPUs are typically employed in a host-device manner via graphics APIs such as DirectX, Vulkan, CUDA, and Metal. The host-side (CPU) applications take responsibility for managing the runtime resources (buffers, textures, etc.) and scheduling the device tasks (memory transfer, kernel execution, etc.). To hide CPU-GPU communication latency and maximize the computation throughput, it is preferable that applications dispatch commands in a non-blocking, asynchronous manner.

While modern GPGPUs have rich computation capability and general-purpose programmability, achieving optimal hardware utilization and performance still necessitates expertise and meticulous adjustment for the intended application and platform. Optimization requires the consideration of both the device-side kernels and host-side scheduling, such as control flow coherence, register pressure, memory access, command grouping, etc., each with numerous context-dependent strategies and trade-offs. For a comprehensive review of today's optimization techniques for GPGPU programming, we refer our readers to [Hijma et al. 2023].

2.2 Domain-Specific Graphics Frameworks

To ease graphics programming, domain-specific languages and frameworks have surfaced since the very inception of programmable graphics hardware. In recent years, with the emergence of ray-tracing hardware, intricate graphics tasks such as physically based (differentiable) rendering are increasingly offloaded onto GPUs. This trend has spurred the development of GPU ray-tracing frameworks to handle highly complicated device-side logic that involves dynamism and polymorphism, as needed by material evaluation.

For example, Shader Components [He et al. 2017] and Slang [He et al. 2018] introduce interfaces and generics into shading languages for maintainable and extensible shader variant management. SLANG.D [Bangaru et al. 2023] brings first-class automatic differentiation to Slang while preserving the type-safe interface feature. Rodent [Pérard-Gayot et al. 2019] exploits AnyDSL [Leißa et al. 2018], a partial-evaluation language and compiler, to generate specialized renderers for each scene to achieve the optimal runtime rendering performance. The Selos [Seitz et al. 2019] shading system employs Terra [DeVito et al. 2013], a multi-stage metaprogramming language, to ease shader implementation. Enoki [Jakob 2019] is the underlying differentiable computing library for the Mitsuba 2 [Nimier-David et al. 2019] renderer. It traces array operations and *just-in-time* (JIT) compiles them into kernels that materialize the computation. Dr.JIT [Jakob et al. 2022] fuses the program traces into mega-kernels, resolving Enoki's performance issues from frequent global memory I/O of intermediate arrays.

Our proposed GPU coroutine model and Dr.JIT share the same objective of generating kernels with appropriate granularity, but the approaches are opposite: Dr.JIT fuses small array operations into larger kernels to reduce launch overhead and I/O traffic, while we split mega-kernels to lower register pressure and improve scheduling. Our model provides better support for control flows: Dr.JIT's symbolic mode does not support kernel splitting and synchronization within control flows, while its evaluated mode tends to incur non-negligible performance penalties.

Luisa [Zheng et al. 2022] combines the idea of embedded domain-specific languages and JIT compilation to enable the dynamic construction of polymorphic shader code. Additionally, it provides a unified runtime layer for cross-platform GPU programming. Our implementation extends Luisa's DSL and runtime interfaces, leveraging its multi-stage programming ability. In Appendix A, we offer a brief review of Luisa to provide readers with an overview of the base system's programming interfaces and capabilities.

In the evolution of these systems, the increasing utilization of compiler knowledge and techniques, such as code transformation and static analysis, is noteworthy. This motivates us to seek the programmatic solution to flexible kernel splitting and scheduling. Indeed, our GPU coroutine implementation also makes heavy use of compiler techniques.

2.3 Coroutines

In the general context, *coroutines* refer to suspendable functions that (1) have persistent program states (values of local data) across successive invocations and (2) allow control reentrance to the point where the control previously leaves at suspension [Marlin 1980].

The coroutine concept has a deep-rooted history since the early era of computer programming languages. According to Moura and Ierusalimschy [2009], Conway [1963] is attributed as the first to introduce coroutines in written literature. During the development of the COBOL compiler, he modeled coroutines as cooperative, intercommunicating subroutines that form an autonomous program together, whose original application was for organizing the

```
Kernel2D kernel = [&](UInt sample_id) {
  Float3 Li = make_float3(0.f);
  Float3 beta = make_float3(1.f);
  UInt2 pixel_id = dispatch_id().xy();
  sampler->start(pixel_id, sample_id);
  auto ray = camera->generate_ray(sampler, pixel_id);
  $for (depth, max_depth) {
    auto hit = scene->intersect(ray);
    $if (hit->miss()) {
      Li += light_sampler()->evaluate_miss(...);
      $break;
    };
    $if (hit->is_light()) {
      Li += light_sampler()->evaluate_light(...);
    };
    auto light_sample = light_sampler()->sample(...);
    auto surf_eval = scene->evaluate_surface(...);
    beta *= ...;
  };
  film->accumulate(pixel_id, Li);
};

Shader2D s = device.compile(kernel);
for(uint i = 0; i < spp; i++) {
  stream << s(i).dispatch(width, height);
}
stream.synchronize();
```

```
// Step 1: add suspension marks to the original kernel
Coroutine coro = [&](UInt sample_id) {
  Float3 Li = make_float3(0.f);
  Float3 beta = make_float3(1.f);
  UInt2 pixel_id = dispatch_id().xy();
  sampler->start(pixel_id, sample_id);
  auto ray = camera->generate_ray(sampler, pixel_id);
  $for (depth, max_depth) {
    $suspend();
    auto hit = scene->intersect(ray);
    $if (hit->miss()) {
      $suspend();
      Li += light_sampler()->evaluate_miss(...);
      $break;
    };
    $if (hit->is_light()) {
      Li += light_sampler()->evaluate_light(...);
    };
    $suspend();
    auto light_sample = light_sampler()->sample(...);
    $suspend();
    auto surf_eval = scene->evaluate_surface(...);
    beta *= ...;
  };
  film->accumulate(pixel_id, Li);
};

// Step 2: create a scheduler to launch tasks
PersistentThreadsSchedulerConfig config{
  .thread_count = 32 * 1024, /* 2^15 */
  .block_size = 128,
};
PersistentThreadsScheduler s{device, coro, config};
// or StateMachineScheduler / WavefrontScheduler
for(uint i = 0; i < spp; i++) {
  stream << s(i).dispatch(width, height);
}
stream.synchronize();
```

Fig. 3. A typical use case of rewriting a mega-kernel path tracer (left) using GPU coroutines and scheduling it with the built-in scheduler (right). Step 1: Users can conveniently specify the splitting points using the `$suspend` keyword, and the system automatically handles the code transformation and live state computation. Step 2: Multiple built-in schedulers, such as the persistent-threads model, are available for scheduling the execution and managing the states of massively parallel coroutine instances on the GPU device.

compilation pipeline of lexer/parser components. Later the coroutine concept was reified in multiple early languages (such as Simula [Birtwhistle et al. 1979] and Modula-2 [Wirth 1988]), with its flexibility in control representation honored. In many modern languages, implicitly or explicitly, from scratch or with extensions, coroutines have become the backbone of many asynchronous programming patterns [Elizarov et al. 2021], such as the `async/await` idiom in C#, Kotlin, Python, Rust, C++, etc.

Despite the diversity in syntactic forms and compiler implementations, coroutines in different languages can be classified by three characteristic dimensions [Moura and Ierusalimschy 2009]:

(1) *Symmetric* vs *asymmetric* control transfer. Symmetric coroutines can transfer control directly to another coroutine, while asymmetric ones may only transfer control to the caller.
(2) *Stackful* vs *stackless* suspension. Stackful coroutines can suspend from inside arbitrarily nested functions, while stackless ones may only suspend from the top-level coroutine body. Stackful coroutines are generally more powerful.
(3) *First-class* vs *constrained* language constructs. Coroutines as first-class language constructs provide users with full control over a coroutine's creation, suspension, resumption, storage, scheduling, etc., while the constrained ones only support restricted forms.

While coroutine support has become common in CPU languages, to our knowledge, there has been little effort to *explicitly* integrate them into GPU kernel programming. This gap may be due to the traditional focus of GPGPUs on data-oriented applications, as coroutines are typically asynchronous constructs used for low-computation, I/O-heavy tasks.

In this context, our exploration of coroutines in a massively parallel environment not only benefits the GPU domain but also provides a complementary perspective to modern CPU languages. Inspired by our work, CPU programmers may consider extending coroutines to schedule data-oriented tasks similarly, such as in entity-component systems in game engines.

2.4 Kernel Reorganization for Scheduling

Some works have investigated kernel or shader reorganization for more efficient scheduling. These efforts can be seen as preliminary explorations that have inspired our GPU coroutine model.

For example, to virtualize the very limited resources on early programmable graphics hardware, Chan et al. [2002] presented a *Recursive Dominator Split* algorithm to partition a fragment shader into multiple passes based on a cost model. This method was then extended by Foley et al. [2004] with support for multiple render targets. Larrabee [Seiler et al. 2008] models and schedules graphics pipeline stages as fibers (i.e., user-space threads) for latency hiding. Attributed to its special many-core x86 architecture with almost identical programmability to a CPU, scheduling is fully performed in a software-based manner with maximized flexibility. OptiX [Parker et al. 2010] implements a specialized compilation process to combine user-provided shaders into a ray tracing mega-kernel, which utilizes a continuation mechanism to pass program states between stages. Recent generations of GPU hardware have introduced *Shader Execution Reordering* techniques to on-the-fly sort threads on the same stream-multiprocessor for better execution coherence [Rusch and Hart 2022]. *GPU Work Graphs* [Patel and Riddell 2024] are another forthcoming feature in DirectX and Vulkan for more efficient and flexible GPU-driven task scheduling.

These works commonly suggest that significant performance improvements can be achieved through kernel splitting and scheduling strategies. However, their implementations are tied to specific architectures or pipelines, limiting their extensibility to other hardware and applications. We abstract and generalize the key insights into an explicit coroutine model, associating kernel splitting with coroutine suspension and sub-kernel scheduling with coroutine schedulers. As we will discuss in Sec. 3, our coroutine model

- is compatible with most current graphics hardware, requiring no specialized architectures or instructions;
- can be easily integrated into existing applications via user-friendly interfaces, necessitating minimal changes;
- decouples the splitting and scheduling steps to allow convenient trials over different combinations of suspension points and schedulers; and
- supports extension with custom schedulers or external rendering/computation passes for novel application scenarios.

This approach is aligned with our goal to provide a generic construct for flexible splitting and scheduling of rendering tasks.

3 DESIGN OVERVIEW

3.1 Objectives and Principles

Coroutines in CPU programming languages have exhibited various forms of design and implementation, each adapted to different languages, tasks, and runtime environments.

As we introduce coroutines into the GPU domain, it is crucial that they maintain a simple yet expressive interface while being tailored to the unique architectures of GPUs. To achieve these objectives, we have defined three key principles for our GPU coroutine model: *compatibility* with current systems, *flexibility* for performance tuning, and *customizability* for novel applications.

Compatibility with current systems. In GPU programming, low-level instructions for function stack management and thread scheduling are typically not exposed to users. As a result, a user-space coroutine implementation, including both language and runtime support, is crucial for ensuring compatibility. Meanwhile, our coroutine constructs are based on the work done in Luisa, which provides a unified abstraction for kernel authoring and runtime management across various platforms. Therefore, our implementation must integrate seamlessly with the existing system components, enabling easy adoption by existing applications with minimal modifications.

Flexibility for performance tuning. Introducing coroutines to GPUs adds another layer of asynchronicity to the already asynchronous task execution. Depending on the workload, there may be multiple ways to designate suspension points within a kernel. The resulting sub-stages can then be scheduled in various ways: either within a reformed kernel, across different kernels, or using a mixed approach. The ability to independently combine splitting and scheduling schemes facilitates application-specific performance tuning.

Customizability for novel applications. While we provide generic schedulers that should perform reasonably well for typical cases, capable users might occasionally want to further optimize their applications with custom schedulers that incorporate domain-specific knowledge. Therefore, besides offering high-level user-friendly interfaces, our framework should also provide access to low-level compiler transformation information and explicit control over subroutine resumption. This will enable novel scheduling schemes and application scenarios that use GPU coroutines as a generic control-flow restructuring construct.

3.2 Choice of the GPU Coroutine Model

The practical design of coroutine interfaces involves two essential components: (1) the syntactic features for declaring coroutines, referred to as the *coroutine language model*, and (2) the runtime constructs for executing coroutines and managing their contexts, known as *coroutine schedulers*.

3.2.1 The language model. We have opted for an *asymmetric*, *stackless* coroutine language model, aligned with those of many modern languages, such as Kotlin [Elizarov et al. 2021] and C++20 [Mazières 2021]. The rationales for this choice are as follows:

- Asymmetric coroutines are considered equally expressive to symmetric ones but have a more structured control hierarchy that simplifies compiler implementation; and
- Stackless coroutines are realizable with pure language-level transformation without dependence on special machine/OS instructions (arbitrary jumps, runtime call stack dumping and restoring, etc.) that are unavailable on GPUs.

Theoretically, this model may be reinterpreted as a *continuation-passing-style* (CPS) transformation on the original program [Appel 1992; Danvy and Filinski 1992; Sussman and Steele 1998]; although in practice, it is usually unnecessary to construct the compiler in an explicit CPS form [Flanagan et al. 1993].

As depicted in Fig. 4, the suspension of a coroutine's execution can be viewed as a *call with current continuation*. This call transfers a closure to the scheduler, subsuming the remainder of the program instructions (i.e., the current continuation) and the program context that carries live states. A later invocation of the continuation closure executes the ensuing instructions from the suspension point as if the original program were resuming.

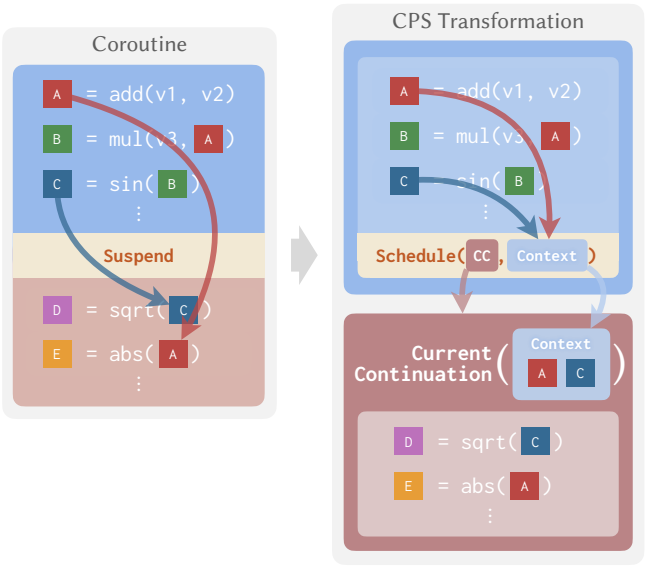


Fig. 4. The *continuation-passing-style* (CPS) transformation on a coroutine. A suspension can be viewed as a *call with current continuation* to the scheduler. The continuation closure contains both the remaining program logic and the live states. A later call to this closure executes the continuation of the suspension point as if the program resumes.

The control over a function's suspension and resumption makes the coroutine model a capable infrastructure for restructuring control flows. As we will demonstrate, many high-level asynchronous programming patterns, such as `async/await`, can be implemented using it. This model also provides enough flexibility for reorganizing GPU kernels, which fulfills the primary purpose of this paper.

3.2.2 The schedulers. Schedulers manage coroutine contexts and orchestrate their suspendable execution. They can be implemented in various styles, existing either as implicit or explicit objects to handle individual or batched coroutine instances. For example, *generators* in Python [Van Rossum et al. 2007] can be viewed as a form of implicit scheduling for individual coroutines. In contrast, coroutines in system languages like C++20 may be scheduled more explicitly with *promise objects* and *coroutine handles* [Mazières 2021], emphasizing versatility and performance.

To strike a balance between ease of programming and flexibility, we expose coroutine schedulers as explicit objects with layered interfaces. For typical use cases, we offer several optimized built-in schedulers with a uniform, high-level interface, allowing users to quickly experiment with various kernel splitting and scheduling schemes. For advanced users, we provide low-level interfaces for direct access to coroutine frames and continuation functions, thus enabling the development of custom schedulers for specific application needs.

4 PROGRAMMING INTERFACES

Following the design of the language model and schedulers, we have extended Luisa's DSL and runtime (see Appendix A for a brief review) to incorporate coroutine support for GPU programming. Fig. 3 shows a typical usage example of how the resulting programming interfaces can be used to split and schedule a path tracer originally written in mega-kernel style.

Specifically, only two steps are required: (1) define the `Coroutine` DSL function with user-designated suspension marks, and (2) create and configure a coroutine scheduler to manage the state storage and execution planning. The first step corresponds to the coroutine language constructs described in Sec. 4.1, while the second pertains to the scheduler interfaces detailed in Sec. 4.2.

4.1 Language Constructs

We have extended Luisa's C++-embedded DSL to incorporate language features for defining and using asymmetric, stackless coroutines. A new function category, `Coroutine`, has been introduced to type the coroutine functions, alongside the existing `Kernel`s and `Callable`s. This extension includes companion support for suspension marks (with the `$suspend` keyword) and high-level programming patterns such as `Generator`s (with the `$yield` keyword) and `$await`, which facilitate coroutine construction and usage. Additionally, for scheduler implementations and advanced applications, low-level interfaces for direct subroutine invocation and state frame management are available on demand.

4.1.1 Coroutine functions. We use a template class type, `Coroutine`, to define device-side coroutines in the DSL:

```
template<typename... Args>
class Coroutine;
```

Similar to `Kernel`s and `Callable`s, the class constructor accepts an underlying definition function of the `Coroutine`, which is invoked with proxy objects to convert expressions and statements into an abstract syntax tree (AST). Note that with user-defined *class template argument deduction* (CTAD) guides [Horton and Van Weert 2023], the template arguments are automatically deduced from the definition function, eliminating the need for manual specification.

```
Coroutine coro = [](Var<Args> args...) {
  /* body of the coroutine to be tracked into an AST */
};
```

Currently, a `Coroutine`'s return type is restricted to `void`. That said, in advanced cases, users can explicitly install values into the coroutine state frame and obtain them elsewhere, as detailed in Sec. 4.1.4. We provide a built-in `Generator` class (Listing 1) for such needs, with usage similar to Python, where the yielded values are passed through a user-designated field, `__yielded_value`, in the frame. Sec. B.1 provides its implementation with our coroutine interfaces.

Automatic program transformation and analysis passes are then performed to lower the coroutine to normal device-side functions, upon which the original mega-kernel-style function is split and materialized into a set of *entry* and *continuation* subroutines and the actual layout of the state frame is calculated.

Like `Callable`s, a `Coroutine` may call built-in functions (e.g., math functions like `sin` and cos) or other user-defined `Callable`s; and the transformed subroutines are invocable from `Kernel`s and `Callable`s. They also share the common ability to capture literal values and runtime resources directly from the host code in Luisa.

```
Kernel1D kernel = [] {
  Generator<uint(uint)> generate_numbers = [](UInt n) {
    UInt i = 0u;
    $while (i < n) {
      $yield(i);
      i += 1u;
    };
  };
  // use a range-for loop to step over the generator
  for (UInt x : generate_numbers(100u)) {
    device_log("x = {}", x);
  }
  // or equivalently, use a while loop
  auto iter = generate_numbers(100u);
  $while (!iter.update().is_terminated()) {
    UInt x = iter.value();
    device_log("x = {}", x);
  };
};
```

Listing 1. Example usage of the built-in `Generator` class. Sec. B.1 provides the internal implementation.

```
Coroutine nested2 = [](UInt n) {
  $for (i, n) {
    device_log("{}/{}", i, n);
    $suspend();
  };
};
Coroutine nested1 = [&](UInt n) {
  $for (i, n) {
    $await nested2(i);
  };
};
Coroutine top_level = [&]() {
  $await nested1(10u);
};
```

Listing 2. Example usage of the built-in `$await` keyword to support chained suspension in nested coroutine invocations. The internal implementation is included in Sec. B.2.

As a special case, it is legal for a `Coroutine` to call the subroutines of another, as they have become ordinary `Callable`s after transformation. This capability can be leveraged to support high-level abstraction patterns previously only available in CPU languages, such as the built-in `$await` keyword we provide for chained suspension in nested coroutine invocations (Listing 2). Sec. B.2 provides the implementation for this feature.

4.1.2 Coroutine suspension marks. A new DSL control-flow keyword, `$suspend`, is introduced for marking suspension points inside coroutines. The compiler transformation passes will automatically split the original kernel at these marks, each leading to a continuation subroutine. Notably, the `$suspend` marks are fully compatible with *any* other control flows. Users can use them within `$if`, `$while`, and `$switch` bodies, regardless of whether these are deeply nested or contain `$break`, `$continue`, and early `$return` statements. This capability is particularly important for handling intricate rendering logic, such as in path tracing.

4.1.3 Low-level interfaces for subroutine selection and invocation. Internally, each suspension point is assigned a unique identification token, which is then mapped to the corresponding continuation subroutine after transformation. Schedulers can directly index the subroutines using these tokens via the low-level interfaces. Moreover, `$suspend` also accepts an optional string tag for advanced users to identify the subroutines more semantically. Listing 3 provides an example of these two indexing styles.

```
Coroutine coro = [&](...) {
  ...
  // suspend with an anonymous token mark
  // (internally maps to token number 1)
  $suspend();
  ...
  // suspend with a user-defined token mark
  // (internally maps to token number 2)
  $suspend("optional mark");
  ...
};
Callable some_scheduler_or_user_function = [&](...) {
  ...
  // invoke a subroutine with the internal token number
  // (typically done in a general-purpose scheduler)
  coro[1](...);
  ...
  // invoke a subroutine with the user-specified mark
  // (for special, application-related needs)
  coro["optional mark"](...);
  ...
};
```

Listing 3. An example of indexing continuation subroutines with internal tokens or user-specified names. Note that this is intended for flexibility in scheduler implementation or advanced custom applications; it is not required for typical use cases of kernel splitting and scheduling.

The transformed subroutines are ordinary `Callable`s, with their first argument being the coroutine state frame. The remaining arguments are inherited from the source `Coroutine`'s definition. As described in Sec. 5.2, the transitions between subroutines and the corresponding frame fields to be saved and restored are encoded in a graph structure. Fig. 5 illustrates the representation of the transformed subroutines.

With the low-level interfaces, users can exert direct control over when and where specific subroutines are resumed and are thus fully responsible for selecting the correct subroutines on the appropriate occasions; otherwise, undefined behaviors may occur. Listing 4 provides an example of how subroutines are reorganized into a generic state-machine kernel using these low-level interfaces directly. Notably, with Luisa's multi-stage programming capabilities, we can dynamically compose the scheduler kernel based on a runtime-known number of subroutines (Line 20–22).

4.1.4 Low-level interfaces for state frame management. Across the suspension-resumption boundaries, some variables and states (such as the target continuation) must be passed on to ensure a correct program context. We refer to these as *live states*. They are automatically computed with our static program analysis passes (Sec. 5).

In typical use cases, as demonstrated in Fig. 3, state frames are managed by schedulers. Users can treat them as opaque and are not required to explicitly handle their creation, storage, or update. However, for scheduler implementations and custom applications, we provide low-level interfaces that allow for direct inspection and manipulation of the coroutine frames.

```
Coroutine coro = [](UInt n) {
  UInt r = n % 2u;
  $if (r == 0u) {
    $suspend("true branch");
    device_log("{} is even.", n);
  } $else {
    $suspend("false branch");
    device_log("{} is odd.", n);
  };
  device_log("{} % 2 = {}", n, r);
};
```

```
struct Frame {
  uint3 coro_id;                    // user-defined coroutine index
  uint target_token;                // token of the target subroutine
  uint r;                           // live state across suspension
};
__device__ void subroutine_entry(Frame &frame, uint n) {
  uint r = n % 2u;
  if (r == 0) {
    frame.r = r;                    // save the live state
    frame.target_token = 1u;        // update the target token
    return;                         // transfer control back to caller
  } else {
    frame.r = r;                    // save the live state
    frame.target_token = 2u;        // update the target token
    return;                         // transfer control back to caller
  }
}
__device__ void subroutine_true_branch(Frame &f, uint n) {
  uint r = frame.r;                 // restore the live state
  device_log("{} is even.", n);     // continuation instruction
  device_log("{} % 2 = {}", n, r);  // continuation instruction
  frame.target_token = TERMINATION; // termination reached
  return;                           // transfer control back to caller
}
__device__ void subroutine_false_branch(Frame &frame, uint n) {
  uint r = frame.r;                 // restore the live state
  device_log("{} is odd.", n);      // continuation instruction
  device_log("{} % 2 = {}", n, r);  // continuation instruction
  frame.target_token = TERMINATION; // termination reached
  return;                           // transfer control back to caller
}
```

Fig. 5. An illustration of the internal representation of the state frame and extracted subroutines of a transformed `Coroutine` after compilation passes. Each subroutine is an ordinary Luisa `Callable`, where the first argument is the state frame and the remaining arguments are inherited from the original definition. Upon resumption, the subroutine reads the live states from the frame and executes the continuation instructions. When the next suspension point or termination is reached, the subroutine saves the live states and updates the `target_token` field of the frame.

```
// definition of a coroutine function
Coroutine c = [&](Args... args) { /* ... */ };

// The subroutines and the state frame are materialized
// after the coroutine transformation passes (done in
// the constructor). Now, the state frame type and the
// subroutines may be used in other device functions.
Kernel3D state_machine = [&](...) {
  // initialize the state frame
  CoroFrame frame = c.instantiate(dispatch_id());
  // call the entry subroutine
  c.entry()(frame, ...);
  // state machine to walk between subroutines
  $loop {
    $switch (frame.target_token) {
      // resume the continuation subroutine associated
      // with the target token of the suspension point
      // note: entry with token 0 is handled aforehand
      //       and is thereby skipped here
      for (uint i = 1; i < c.subroutine_count(); i++) {
        $case (i) { c[i](frame, ...); };
      }
      // terminate if no continuation
      $default { $return(); };
    };
  };
};
```

Listing 4. An example of reorganizing the transformed subroutines into a state-machine kernel, utilizing the low-level coroutine frame management and subroutine invocation interfaces directly.

We introduce a built-in `CoroFrame` type for in-kernel state frame management with the DSL, similar to the existing `Var<T>` objects. The class definition is shown in Listing 5, where the `CoroFrameDesc` object, provided by the compiler passes, describes the underlying coroutine frame structure and the name-to-index mapping of user-designated fields. The `CoroFrame` utilizes this descriptor to create internal DSL variables, referenced by the `RefExpr` member.

The underlying representation of the frame is a standard Luisa structure. For compact storage, live states are decomposed into scalar fields and packed into a single frame for all subroutines from the same coroutine.

The dispatch index (`coro_id`) and the token of the target continuation (`target_token`) are two special fields in the frame. These fields are utilized by schedulers to identify the location and execution state of a coroutine instance. The dispatch index is (optionally) initialized at the frame's creation during coroutine instantiation, and the target token is automatically updated at each suspension point.

```
// dispatch_id() implicitly reads coro_id in coroutines
Coroutine coro = [](...) { auto id = dispatch_id(); };
// create a frame with the user-provided index
CoroFrame frame = coro.instantiate(index);
// read the stored index explicitly
UInt3 index = frame.coro_id;
// read the token of the resumption target
UInt target = frame.target_token;
```

We also provide methods for explicitly storing a DSL value as a state frame field within a `Coroutine` using the new keyword `$promise`.

```
class CoroFrame {
private:
  const CoroFrameDesc *_desc; // frame descriptor
  const RefExpr *_expression; // variable in the AST

public:
  // create a frame variable with the given descriptor
  static CoroFrame create(const CoroFrameDesc *desc);

public:
  // special field for the dispatch index
  UInt3 &coro_id;
  // special field for the target subroutine token
  UInt  &target_token;
  // method to retrieve a field at the given index
  template<typename T> Var<T> &get(uint index);
  // method to retrieve a user-designated field
  template<typename T> Var<T> &get(string_view name);
};
```

Listing 5. Definition of the `CoroFrame` class for direct management of coroutine frames in the DSL. The actual layout of the underlying structure is automatically computed by the compiler analysis passes. In typical use cases, coroutine frames are managed by schedulers and opaque to users.

This value can later be retrieved from the frame by name in other functions using the `CoroFrame::get<T>(string_view)` method:

```
Coroutine coro = [](...) {
  $promise("some_value", x); // designate a field
};
// read the designated frame field (in other functions)
UInt v = frame.get<uint>("some_value");
```

This feature facilitates the implementation of `Generator`s (Sec. B.1) and simplifies the interaction with external custom passes.

We have specialized the buffer and shared memory class templates in Luisa, enabling a frame to be easily stored in thread-local, block-shared, or global memory. This flexibility allows for various data layouts, including SoA (Structure-of-Arrays) and AoS (Array-of-Structures), as well as other feasible configurations.

4.2 Scheduler Interfaces

Coroutine schedulers orchestrate the actual execution and resource management. We offer several built-in schedulers for user convenience. All these schedulers derive from the same base abstract class, as shown in Listing 6. This setup enables convenient experimentation with different schedulers for the same coroutine.

Specifically, derived schedulers are required to implement the virtual method `_dispatch`, which creates and dispatches the specified number of coroutine instances to the command stream. This process uses the provided coroutine arguments and may occur in a single pass or multiple passes. Necessary information, such as the subroutine transition graph and the coroutine frame descriptor, is accessible through the stored `Coroutine` object.

Schedulers have complete freedom to organize the transformed subroutines into kernels, manage coroutine frames, and coordinate command submission and execution. This flexibility allows for various scheduling schemes, including the built-in ones and their diverse options that we provide. For instance, Listing 7 illustrates how a naive state-machine scheduler can implement this abstract interface.

```
template<typename... T>
class CoroScheduler {
protected:
  Coroutine<T...> _coro;

private:
  virtual void
  _dispatch(Stream &stream, uint3 size, T... args) = 0;

public:
  CoroScheduler(Coroutine<T...> coro) : _coro{coro} {}
  // for syntactic sugar:
  //   stream << sched(args...).dispatch(size)
  // which internally calls the _dispatch() method
  CoroTaskSubmitter operator()(T... args) { /*...*/ }
};
```

Listing 6. The abstract interface for our built-in coroutine schedulers.

```
template<typename... T>
class StateMachine : public CoroScheduler<T...> {
private:
  Shader3D<T...> _shader;

private:
  void _dispatch(Stream &stream, uint3 size, T... args)
  override {
    stream << _shader(args...).dispatch(size);
  }

public:
  StateMachine(Device &device, Coroutine<T...> coro)
    : CoroScheduler{coro} {
    Kernel3D state_machine = [&](T... args) {
      /* see definition in Listing 4 */
    };
    _shader = device.compile(state_machine);
  }
};
```

Listing 7. Implementation of the naive state-machine scheduler using the abstract scheduler interface defined in Listing 6.

We will provide further details on the implementation of the built-in schedulers in Sec. 6.

If beneficial, users can opt to implement customized scheduling schemes in addition to the built-in ones, allowing application-specific knowledge to be leveraged together with the rich information we expose. One example is provided in Sec. 7.3, where an external finite-difference pass is inserted at the suspension point to estimate the ray differentials before texture evaluation. Additionally, multiple common utility classes and functions are provided to facilitate the implementation of new schedulers, such as task queues with atomic counters and device-side sorting kernels.

4.3 Comparison with the Alternative Model

Readers might wonder why we chose to expose explicit suspension marks, along with supplemental low-level management interfaces for advanced usage. An alternative approach could have solely offered the more established `async/await` constructs, delegating all scheduling to the compiler, as is common in many CPU languages. However, we found our decision necessary to guarantee the desired flexibility and performance when scaling to GPU programming:

- Our model handles massively parallel GPU threads rather than a few standalone coroutine instances. Efficient scheduling requires fine-grained control over the frame layout and the ability to reorganize subroutines. The `async/await` model would be too opaque for implementing schedulers.
- The `$suspend` keyword marks the splitting point of the original kernel. Making it explicit allows users to experiment with different splitting strategies without altering the logic.
- We target the conventional kernel-based GPU programming model, not the array-based style where each operation is itself asynchronous. Rewriting a mega-kernel with complicated logic, such as a path tracer, into `async` function chains requires significant control-flow restructuring, which is non-trivial, especially when handling nested branches and loops.
- High-level patterns such as `await` and generators can be constructed atop the low-level interfaces, but not vice versa.

Admittedly, this preference for flexibility and performance does come at a cost. When directly operating the low-level interfaces, it is the developers' responsibility to ensure (1) the appropriate creation, storage, and recovery of coroutine frames and (2) the correct choice and invocation of the expected continuation subroutines with the corresponding frames. Otherwise, undefined behavior might occur. However, such safety constraints mostly pertain to advanced usage cases, such as custom scheduling. For typical users, using the built-in schedulers is generally sufficient and recommended, as demonstrated by the path tracing example in Fig. 3.

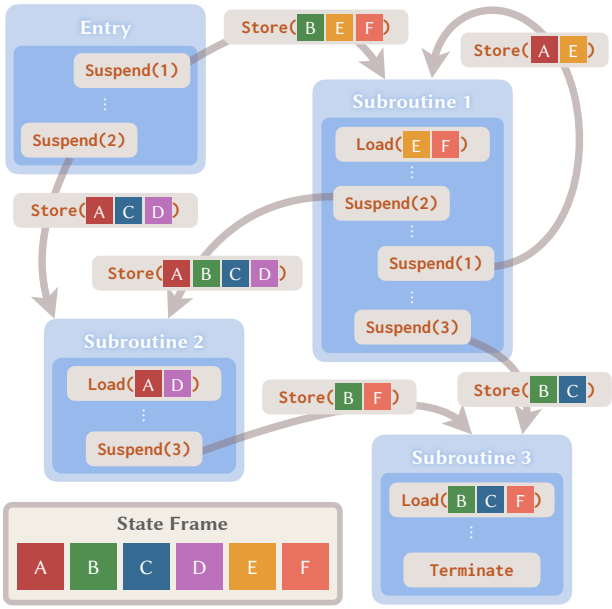


Fig. 6. An example coroutine transition graph. Each node in the graph represents a continuation subroutine. The suspension-resumption relations between them are modeled as directed edges with field-wise usage information of the state frame. The states to load on subroutine resumption and to store on suspension are also recorded in the graph.

5 LANGUAGE FEATURE IMPLEMENTATION

To implement the language features for the coroutine model, we perform a set of compiler transformations and analysis passes to lower the DSL `Coroutine` constructs to ordinary device-compatible functions and data types in Luisa.

5.1 Subroutine and State Frame Extraction

As mentioned in Sec. 3.2, our coroutines are equivalent to the CPS transformation on the original program: a coroutine suspension transfers the current continuation and the program context to the scheduler, so when the scheduler calls back the continuation with context, instructions in succession of the suspension point are executed as if the original program resumes.

From this view, the substantial objective of our implementation is to solve two problems for each suspension point:

(1) Which instructions would be executed in the continuation?
(2) What context should be passed on to the continuation?

Their solutions are supplied by the two major stages of the code transformation passes: *continuation extraction* and *state frame materialization*. Besides, auxiliary pre and postprocessing passes are also employed to canonicalize the input programs and organize the transformed programs into a frontend-friendly form, respectively.

Details of the compiler analysis and transformation passes can be found in Appendix C. Although the implementation follows a similar outline to that of existing compilers, such as LLVM [Lattner and Adve 2004] and Kotlin [Elizarov et al. 2021], special attention must be paid to the critical per-thread resource efficiency when targeting massively parallel GPUs. We combine intra-scope use-define analysis (Sec. C.3.1) and inter-scope liveness analysis (Sec. C.3.2) on a per-aggregate-member basis to compute the involved live states at each suspension-resumption boundary and pack them into a compact state frame. The analysis results are fed back to the frontend via the transition graph to aid scheduler implementation.

Besides, to accommodate the source-to-source transformation within Luisa's structured IR, we perform a control flow normalization pass in the preprocessing step (Sec. C.1.1) and develop a condition replay strategy to reconstruct the control flow of the extracted continuations (Sec. C.2.3).

5.2 Subroutine Transition Graph Construction

The products of the compiler passes — the split subroutines and the state transitions between them — are encoded into graphs. Fig. 6 depicts the structure of an example coroutine state transition graph.

Specifically, the split subroutines, each associated with an entry or a suspension point, are represented as graph nodes. They are materialized into normal device functions (i.e., `Callable`s in Luisa) that manipulate the live state frame and can be invoked both from kernels and other device functions, allowing users to conveniently wrap them into various scheduler styles.

Continuation relations between the subroutines are modeled as graph edges, each connecting the source and target subroutines of a suspension-resumption pair. We also embed field-wise usage information (from the coroutine transformation) in the state frame for each edge, detailing which fields should be written to at suspension and which are read upon resumption. Such information helps

Table 1. A characteristic comparison between the built-in coroutine schedulers. The table lists the general performance concerns and overhead sources in GPU programming, where the "Low" ones are desired while the "High" ones are better to avoid.

Scheduler	Naïve	Wavefront	Persistent
Scheduling Granularity	Thread	Device	Block
Thread Divergence	High	Low	Low
Register Pressure	High	Low	High
Global Memory Traffic	Low	High	Low
Scheduling Overhead	Low	High	High

reduce memory traffic, especially when utilized in a wavefront-style scheduler with the SoA layout.

Schedulers might exploit the subroutine transition graph to optimize the execution of the coroutines.

5.3 Support for High-Level Asynchronous Patterns

Like in many CPU languages, we support `Generator`s and the `$await` operator to facilitate structured asynchronous programming with coroutines. As detailed in Appendix B, they are implemented through the coroutine interfaces described in Sec. 4.1. This demonstrates the expressiveness of our GPU coroutine model and the flexibility of the provided interfaces. Other patterns are similarly possible.

6 BUILT-IN SCHEDULER IMPLEMENTATION

The coroutine transformation splits a mega-kernel into smaller subroutines, each carrying the continuation at the corresponding suspension point. Calling a subroutine with the state frame effectively runs the continuation with the stored context, as if the original program resumes. When reaching the next suspension point (or the final termination), the subroutine updates the state frame and transfers control back to the caller.

Therefore, the design space of a coroutine scheduler includes two major dimensions:

(1) The management of the coroutine state frames; and
(2) The arrangement of the subroutine execution occasions.

Modern GPUs provide quite rich options for both dimensions. For example, the state frame can be stored locally, in the block-shared memory, in the global memory, and so forth; either with the layout of array-of-structures (AoS), structure-of-arrays (SoA), or even their mixtures. Subroutines can be one-to-one wrapped into kernels or selectively grouped, with each kernel executed by a fixed thread or dynamically acquired by a persistent thread, etc. Davidovič et al. [2014] reviewed various kernel organization and scheduling strategies for GPU renderer implementation.

We provide three representative built-in schedulers out of the numerous choices for user convenience. They are in the state-machine, wavefront, and persistent-threads styles, respectively. Table 1 compares their characteristics. We hope they can act as good starting points for users to test different coroutine splitting schemes and develop their application-specific scheduling strategies.

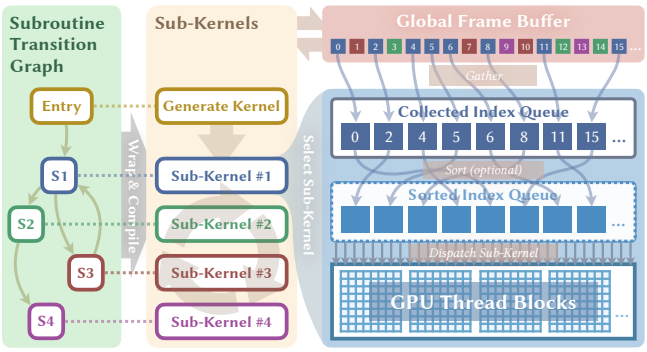


Fig. 7. The design of our wavefront coroutine scheduler.

6.1 Naïve State-Machine Scheduler

The simplest way to run the subroutines is to reassemble them back into a mega-kernel state machine. The aforementioned example code in Listing 7 already provides the implementation.

This scheduler one-to-one maps a coroutine instance to a GPU worker thread. A thread-local state frame is created first and the entry subroutine is called. Then the kernel starts the state machine, repeatedly polling the target token to determine and invoke the associated subroutine until the special termination token is encountered. Note that the entry subroutine may not be the continuation of any subroutine, thus called only once at the beginning and never included in the state machine.

The forth-and-back conversion seems rather redundant. The mega-kernel merely cycles its way back to yet another form of mega-kernel without any possible performance or flexibility gains from the coroutine transformation. However, its extreme simplicity serves well as a testbed to validate the coroutine transformation and a performance baseline in the experiments.

6.2 Wavefront Scheduler

The wavefront scheduler is a straightforward generalization of the wavefront path tracing technique. As shown in Fig. 7, the basic idea is to decompose the state machine into multiple sub-kernels, each handling a subroutine. The state frames are stored in a global buffer, loaded by the subroutines on resumption, and updated on suspension. Transitions between subroutines are tracked by the target token field in the frame.

In each iteration, we check the global frame buffer and collect the indices of alive instances into a queue. A *stable multi-split* algorithm [Ashkiani et al. 2016] is employed to cluster the indices by their corresponding subroutines, ensuring physical coherence and reducing the scheduling overhead. The sub-kernels are then dispatched to process their attached instances referenced in the queue. When a coroutine reaches termination, a new instance (if any more) can be generated in place at the same frame buffer slot.

In this way, control flows are more likely to converge since all threads execute the same subroutine during a dispatch. Register pressure is also alleviated, with each sub-kernel containing only a portion of the computation from the original mega-kernel and hence fewer temporary states to keep.

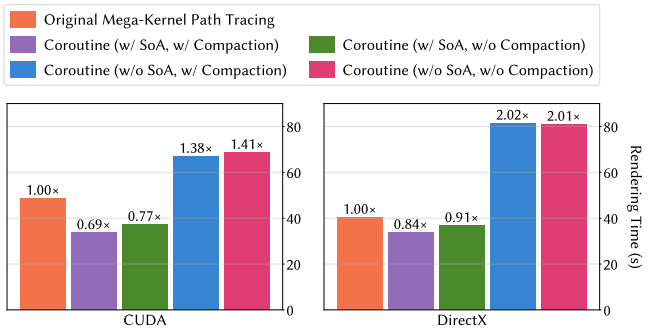


Fig. 8. Ablation study of the *SoA frame layout* and *frame buffer compaction* optimizations in the wavefront coroutine scheduler. The figure shows the rendering time (in seconds, shorter is better) of two backends: CUDA and DirectX. The comparison is performed on the *Lone Monk* scene rendered at 1920x1440/1024spp, with 10 maximum bounces and Russian Roulette (RR) applied from the 2nd bounce, on RTX-2080Ti. Both the SoA frame layout and frame buffer compaction options help to reduce global memory traffic, improving the overall performance.

Pragmatic details are yet to be arranged, such as the state frame layout and sub-kernel launch order, which can have a significant impact on performance. To improve the adaptation to different scenarios, we provide several tuning options, inspired by existing works [Aila and Laine 2009; Blender Online Community 2024; Garanzha and Loop 2010] on accelerating wavefront path tracing.

SoA frame layout. The SoA layout is a "transposition" of the conventional AoS layout by storing each field of the state frame as a separate scalar buffer. This enables sub-kernels to load and store only the necessary state frame fields with the information from the transition graph, reducing the required memory bandwidth. Also, the scalar-stride layout may better fit the cache units on modern GPUs, which can coalesce adjacent memory accesses into the same cache sectors. However, for random access patterns without much locality, the SoA layout may on the contrary waste the cache sectors.

Frame buffer compaction. Coroutine instances usually have uneven lifetimes and thus terminate at varying times. Gaps are left in the frame buffer when no more new instances are generated for replacement, which hurts the locality of memory access. Users can enable compaction to collect all active instances to the beginning of the buffer when the load factor (percentage of non-terminated coroutine instances) is below some threshold. However, the relocation of large frames can consume a considerable memory bandwidth.

The effects of this option are studied in Fig. 8, together with the aforementioned SoA layout optimization.

6.3 Persistent-Threads Scheduler

The mega-kernel state machines use only thread-local storage but can often suffer from thread divergence and register pressure; while the wavefront scheduler improves thread coherence but at the cost of an undesired increase in global memory traffic, despite the optimization options we support. Modern GPUs organize threads in a block-wise manner on the stream-multiprocessors. Therefore, the

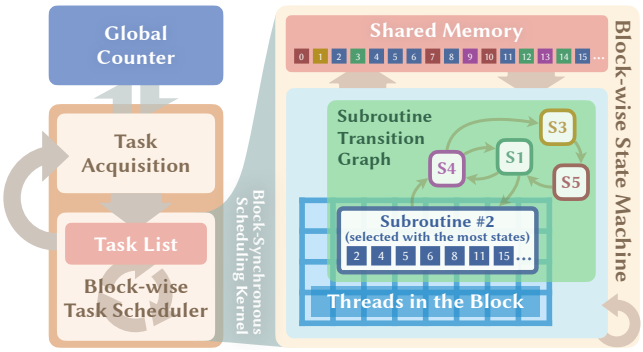


Fig. 9. The design of our persistent-threads coroutine scheduler.

hidden balance between thread coherence and memory traffic might be sought with the block-wise storage and computation resources.

The persistent-threads [Gupta et al. 2012] scheduler attempts to exploit the fast block-shared memory (a programmable low-latency on-chip L1 cache on modern GPUs shared by threads per block) for state frame storage instead of frequent expensive global memory accesses, and run block-wise state machines with batched coroutine instances to improve intra-block thread coherence. Scheduled in this style, threads are no longer transient with the same lifetime as a single task but *persist* over multiple batches of tasks, hence the name. Fig. 9 illustrates our persistent-threads scheduler design.

Task acquisition. A thread block acquires a batch of pending tasks when the block decides to generate new tasks. The request is made by a single atomic addition operation on a global task index counter, issued from the leader thread in the block. To reduce global memory atomic operations and ensure sufficient parallelism, the batch size should be reasonably large (but not too large), typically a few times of the block size.

The thread block then loops over the fetched coroutine tasks, initializing their state frames in the shared memory. Block-wise counters (stored in the shared memory, too) are also updated to track the instance count of each subroutine.

Block-wise state-machine execution. Subroutines are organized into a single-kernel state machine but executed blockwise-synchronously. On each step, the leader thread peeks the subroutine counters to pick the most common continuation in the block. Analogous to the wavefront scheduler, indices of the interested instances are collected into a queue, but stored in the fast shared memory this time. Then, all threads run the same selected subroutine on the collected instances, with the associated frames loaded from the shared memory into local storage before invocation and written back after. Updates on the target token field effectively drive the instances to the next stage or the termination. When all coroutine instances are complete, the block is available again to acquire new work from the global pool.

The block-wise state-machine model replaces the expensive global memory operations and global synchronization in the wavefront scheduler with the low-cost shared memory accesses while achieving comparable thread coherence when configured with a sufficient

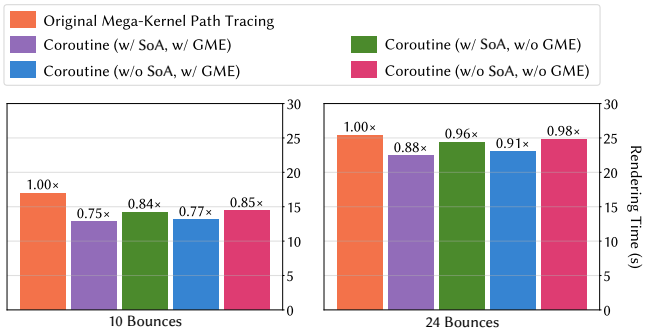


Fig. 10. Ablation study of the *bank conflict avoidance with SoA layout* (denoted as "SoA") and *global memory extension* (denoted as "GME") optimizations in the persistent-threads scheduler. The figure shows the rendering time (in seconds, shorter is better) of two different settings: (left) 10 maximum bounces with Russian Roulette (RR) applied from the 2^{nd} bounce and (right) 24 maximum and RR from the 5^{th}. The comparison is performed on the *Kitchen* scene rendered at 1280x720/1024spp using the DirectX backend. Both optimizations improve parallelism, SoA by avoiding shared memory bank conflicts, and GME by reducing the working set storage.

batch size. However, register pressure can still impede parallelism since subroutines are all arranged into a single kernel.

Bank conflict avoidance with SoA layout. Bank conflicts occur in GPU shared memory when threads within a warp access different addresses in the same memory bank simultaneously. In such cases, the memory requests must be serialized, resulting in performance degradation. To mitigate this, our persistent-threads scheduler can be configured to use the SoA memory layout for state frames, reducing the likelihood of bank overlaps due to large aligned structures. Fig. 10 shows the ablation study results for this option.

Global memory extension. If the persistent-threads implementation preserves all frames only in the shared memory, parallelism would be partly limited by the available shared memory size (typically tens of kilobytes for each block on the latest GPU generations), since just some of the threads in a block might be used in execution.

To sidestep this limitation, we use a global buffer as an extension for state frame storage, which together with the shared memory keeps enough frames to ensure that every launch fills all threads in the whole block. When the shared memory size is not enough to hold all frames for the selected instances in a state machine step, some currently unused ones will be swapped to the global memory. However, frequently accessed fields, such as the coroutine indices, are always independently kept in the shared memory to avoid undesired overhead.

With this extension, we trade small costs in global bandwidth for better thread occupancy. The strategy is especially beneficial when the latency-hiding ability outweighs the increased memory I/O. The effects of this optimization are studied in Fig. 10.

7 DEMONSTRATION APPLICATIONS

To examine the practicality and performance of GPU coroutines, we apply them to three demonstration applications: path tracing (Sec. 7.1), signed distance field (SDF) rendering (Sec. 7.2), and texture filtering as an inserted custom pass during suspension (Sec. 7.3). All the experiments are conducted on RTX-2080Ti (11GB VRAM) with Luisa's DirectX and CUDA backends.

7.1 Path Tracing with Coroutines

Our GPU coroutine can be applied to conveniently split a mega-kernel path tracer into subroutines and schedule them in different styles: users could place suspension marks at proper occasions in the path tracing logic and use a built-in or customized scheduler to launch the rendering.

In our experiments, we fork the mega-kernel path integrator in LuisaRender[1] and suspend the radiance computation at 4 locations: (1) before ray intersection at each bounce; (2) when the ray misses the scene geometry; (3) before sampling the direct lighting; and (4) before surface material evaluation. Other suspension points are also possible but we find that these are generally good choices and also correspond to the logical stages of path tracing. This process effectively splits the mega-kernel into five sub-stages: camera ray generation, ray intersection, miss ray handling, light sampling and occlusion checking, and surface evaluation and sampling. Fig. 3 illustrates this rewriting process.

For scheduling, we use the wavefront and persistent-threads schedulers. The naïve state-machine scheduler is excluded since it gives similar results to the original mega-kernel, as expected. The wavefront scheduler is configured with 2^{24} coroutine instances per kernel dispatch, SoA frame layout, and frame buffer compaction. Effects of the SoA and compaction options are evaluated as in Fig. 8. The persistent-threads scheduler is configured to use 2^{15} working threads, with 128 threads per block each fetching 16 coroutine instances on task acquisition. The bank conflict avoidance and global memory extension optimizations are both turned on.

We test five scenes. *Bathroom*[2], *Salle de Bain*[3], *Kitchen*[4], and *White Room*[5] are from *Rendering Resources* [Bitterli 2016]. *Lone Monk*[6] is converted from [Bergonzini 2021]. All scenes are rendered in spectral mode at 1024spp with two maximum/Russian Roulette (RR) tracing depth settings, on CUDA and DirectX backends. The results are shown in Fig. 11.

In most cases, both schedulers outperform the original mega-kernel version from LuisaRender on the test scenes. Additionally, the wavefront scheduler performs slightly better than the persistent-threads scheduler. This indicates that register pressure and thread divergence might be the dominant bottlenecks in complex rendering tasks like path tracing.

To better understand the performance gains, we profiled the register usage, compute throughput, memory throughput, and branch efficiency of the original mega-kernel and our coroutine-transformed wavefront versions. Table 2 shows the results using the CUDA backend on the *Lone Monk* scene, rendered with a maximum of 10 ray bounces. Our coroutine-transformed wavefront version significantly

[1]We use the open-source version at https://github.com/LuisaGroup/LuisaRender.
[2]Courtesy of Mareck. Licensed under CC0 1.0.
[3]Courtesy of nacimus. Licensed under CC BY 3.0.
[4]Courtesy of Jay-Artist. Licensed under CC BY 3.0.
[5]Courtesy of Jay-Artist. Licensed under CC BY 3.0.
[6]Courtesy of Carlo Bergonzini, Monorender. Licensed under CC0 1.0.

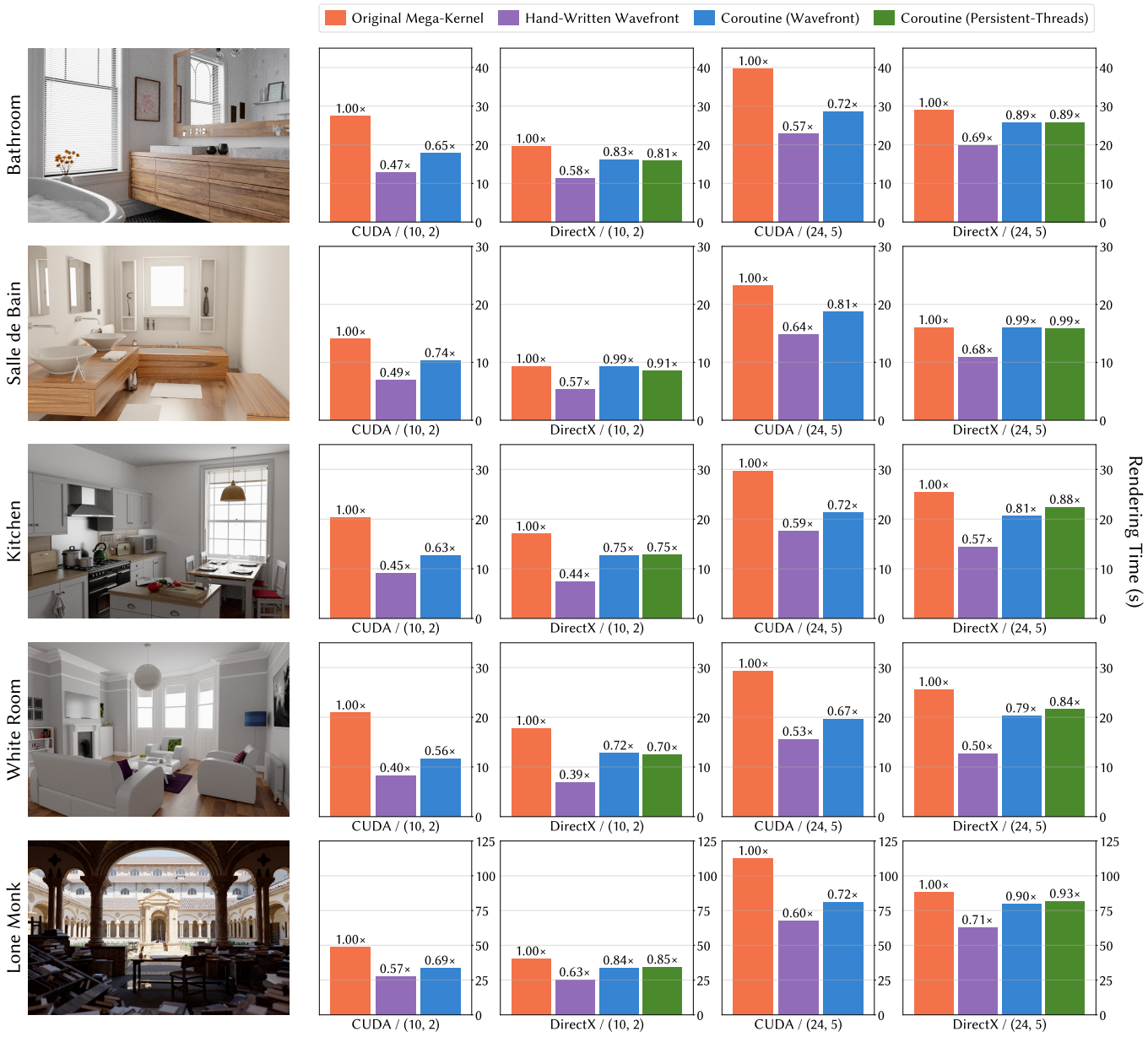


Fig. 11. Rendering time comparison for the path tracing application with GPU coroutines. The figure shows the time (in seconds, shorter is better) to render 5 test scenes on CUDA and DirectX backends, with 10 maximum bounces, Russian Roulette (RR) applied from the 2nd bounce (10, 2) and 24 maximum bounces, RR applied from the 5th bounce (24, 5). The persistent-threads scheduler is not available on CUDA due to the absent support for thread block synchronization when ray tracing with OptiX. *Bathroom* is rendered at a resolution of 1024x1024; *Salle de Bain*, *Kitchen* and *White Room* at 1280x720; *Lone Monk* at 1920x1440. In most cases, both schedulers outperform the original mega-kernel version from LuisaRender, and the wavefront scheduler performs slightly better than the persistent-threads scheduler. The naïve state-machine scheduler is excluded since it gives similar results to the original mega-kernel, as expected.

reduces register pressure and thread divergence in the intersection and light sampling stages by isolating the most resource-intensive surface evaluation computations into a separate kernel. In contrast, the original mega-kernel's performance is limited by live registers and diverged branches in shading, resulting in overall low hardware utilization.

It is not surprising that our automated solution still exhibits noticeable performance gaps compared to the hand-written wavefront version from LuisaRender. This disparity is partly due to our solution's less mature optimization capabilities. For instance, the hand-written version features a smaller, manually optimized state frame, where some simple values are recomputed rather than stored in

Table 2. Profiling results for the path tracing application comparing the original mega-kernel and coroutine-transformed wavefront versions on the *Lone Monk* scene, rendered using the CUDA backend with a maximum of 10 ray bounces. The coroutine-transformed version shows reduced register pressure and thread divergence by isolating the most resource-intensive surface evaluation into a separate kernel, thereby optimizing the intersection and light sampling stages and improving overall hardware utilization. In contrast, the original mega-kernel's performance is constrained by its low overall hardware utilization.

Implementation	Mega-Kernel	Coroutine (Wavefront)					
		Ray Generation	Intersection	Miss	Light Sampling	Surface Evaluation	Scheduling
Time Proportion	100%	3.0%	19.7%	2.7%	13.2%	51.7%	9.7%
Register Usage	128	64	86	72	82	200	—
Compute Throughput	6.4%	34.2%	4.6%	1.97%	18.7%	6.9%	—
Memory Throughput	23.9%	55.4%	52.1%	45.3%	42.5%	22.7%	—
Branch Efficiency	88.7%	100%	98.34%	100%	97.6%	86.2%	—

the frame to reduce memory traffic. Moreover, the hand-written version leverages knowledge of the path tracing logic to arrange the kernel launch order without needing to read back the path queue sizes. In contrast, our general-purpose scheduler implementation incurs a non-negligible amount of scheduling overhead (e.g., 9.7% of the overall time on the *Lone Monk* scene, as shown in Table 2). Nevertheless, we remain optimistic that continued research and development will eventually bridge the gap between the two.

7.2 SDF Rendering with Coroutines

We also extend our usage of coroutine to SDF rendering. SDF rendering uses iterative ray marching to find intersections with the (usually procedural) geometry. Uneven ray marching iterations with varying numbers of distance evaluations in this procedure often create imbalanced and divergent workloads in nearby pixels inside the same block. Leveraging our automatic coroutine transformation and the built-in schedulers, we can test different optimizing configurations to improve hardware utilization.

We test this idea on an SDF rendering example ported from Taichi [Hu et al. 2019]. A single suspension point is placed before the ray marching process. The wavefront and persistent-threads schedulers are tested and results are displayed in Fig. 12. The persistent-threads scheduler outperforms in this task. This might be due to the relatively simple shading, which does not stress the registers but sensitively exposes the wavefront overhead in memory bandwidth and scheduling. Whereas the persistent-threads scheduler improves the thread coherence and load balance without an analogous overhead.

7.3 Custom Texture Filtering Passes during Suspension

The previous two applications focus on the performance benefits of splitting mega-kernels for optimized scheduling. That said, GPU coroutines in essence are powerful control structures, upon which many programming patterns may be built.

For example, suspension can serve as a barrier, during which the application may swap in another task before the coroutine's deferred resumption. This facilitates the incorporation of custom computational passes, especially those inexpressible in an embarrassingly parallelized manner, inside arbitrarily nested control flows.

We demonstrate this idea on texture filtering in path tracing. Standard point or bilinear texture filtering is prone to aliasing at low sample rates, while trilinear and anisotropic filtering works well but

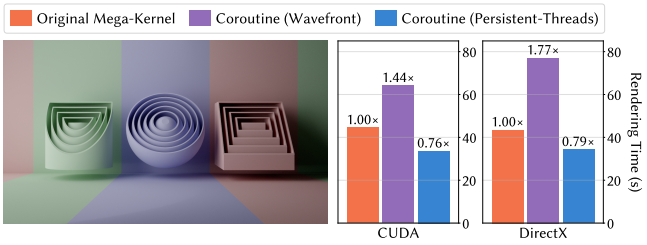
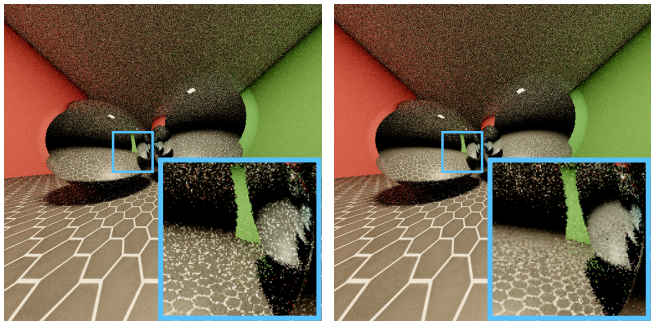


Fig. 12. Rendering time comparison for the SDF rendering application with GPU coroutines. The figure shows the rendering time (in seconds, shorter is better) on two backends, CUDA and DirectX. The comparison is performed on the SDF renderer, rendered at 1280x720/16384spp on RTX-2080Ti. The persistent-threads scheduler outperforms the wavefront scheduler, which might be due to the relatively simple shading.



(a) Bilinear (b) Anisotropic

Fig. 13. An example that incorporates custom texture filtering passes in path tracing with coroutines. Our GPU coroutines can be used as a general control construct that splits device functions at suspension points. This allows the flexible insertion of custom passes during suspension inside arbitrary control flow without manually rewriting the entire kernel. Both images are rendered at 2spp, with (a) standard bilinear texture filtering without ray differentials and (b) anisotropic filtering using a custom finite-difference pass to compute ray differentials during suspension, respectively.

requires tracking of ray differentials. In traditional path tracing, each path is independently computed so a trivial finite-difference-based method (as in rasterization) is inaccessible.

```
// global memory for inter-shader communication
auto uv_img =
  device.create_image<float>(FLOAT2, width, height);
auto surface_id_img =
  device.create_image<uint>(UINT1, width, height);
auto duvdxy_img =
  device.create_image<float>(FLOAT4, width, height);

// define the coroutine for kernel splitting
Coroutine coro = [&](UInt sample_id) {
  Float3 Li = make_float3(0.f);
  Float3 beta = make_float3(1.f);
  UInt2 p = dispatch_id().xy();
  sampler->start(p, sample_id);
  auto ray = camera->generate_ray(sampler, p);
  $for (depth, max_depth) {
    auto hit = scene->intersect(ray);
    /* ... */
    uv_img.write(p, hit.uv);
    surface_id_img.write(p, hit.surface_id);
    $suspend("surface eval");
    hit.duvdxy = duvdxy_img.read(p);
    auto surf_eval = scene->evaluate_surface(hit);
    /* ... */
  };
  film->accumulate(p, Li);
};
// create the coroutine state frame buffer and
// reorganize the subroutines into separate kernels
auto frame_buffer = device.create_buffer<CoroFrame>(
  coro.frame_desc(), width * height);
Kernel2D entry = [&](UInt sample_id) {
  UInt2 p = dispatch_id().xy();
  CoroFrame frame = coro.instantiate(p);
  coro.entry()(frame, sample_id);
  frame_buffer.write(p.y * width + p.x, frame);
};
Kernel2D continuation = [&](UInt sample_id) {
  UInt2 p = dispatch_id().xy();
  UInt i = p.y * width + p.x;
  CoroFrame frame = frame_buffer.read(i);
  $if (!frame.is_terminated()) {
    coro["surface eval"](frame, sample_id);
    frame_buffer.write(i, frame);
  };
};
auto entry_shader = device.compile(entry);
auto cont_shader = device.compile(continuation);
// create an external finite-difference pass to
// estimate ray differentials from adjacent pixels
Kernel2D duvdxy = [&]() {
  UInt2 p = dispatch_id().xy();
  Float4 duvdxy = fd(p, uv_img, surface_id_img);
  duvdxy_image->write(p, duvdxy);
};
auto duvdxy_shader = device.compile(duvdxy);

// execute the shaders with the application-specific
// knowledge of the suspension point and max_depth
for (uint i = 0; i < spp; i++) {
  stream << entry_shader(i).dispatch(width, height);
  for (uint d = 0; d < max_depth; d++) {
    // insert the external pass during the suspension
    stream << duvdxy_shader(i).dispatch(width, height)
           << cont_shader(i).dispatch(width, height);
  }
}
```

Listing 8. An example of incorporating an external finite-difference pass during path tracing, facilitated by coroutines. Coroutines may serve as a versatile control flow manipulation tool, allowing the kernel logic to be partitioned at arbitrary points. This simplifies the insertion of external passes that require global synchronization during embarrassingly parallel execution, such as in the finite-difference process, where the texture coordinates of adjacent pixels are needed.

With coroutines, however, we can suspend the path tracing logic before surface evaluation, with the texture evaluation contexts stored in global memory. An extra finite-difference pass is then launched to estimate the derivatives by differentiating the texture coordinates of adjacent paths in image space, with the results written back to global memory. Upon resumption, the paths use the differentials for anisotropic filtering and continue the rendering. Listing 8 provides the implementation, and Fig. 13 shows the example renderings.

Although this is a basic application, it promisingly showcases the potential of the coroutine concept on GPUs. We anticipate that more sophisticated use cases will arise, e.g., the integration of neural network passes into a computation kernel. We look forward to future research exploiting the full expressive capacity of this model.

8 CONCLUSION

We introduced coroutines into GPU kernel programming as a unified and automated approach to splitting and scheduling intricate rendering tasks. We designed an asymmetric stackless coroutine model suitable for modern GPUs and reified it with programming language support and built-in schedulers by extending the DSL and runtime facilities in Luisa. We applied our GPU coroutine implementation to several demonstration applications, including path tracing, SDF rendering, and interaction with external passes, to showcase its flexibility in task scheduling and control representation.

However, there remains a distance to optimal performance. Due to the limited compiler optimization techniques available in the system, the current implementation may not always yield optimal code transformation and state frame analysis results, leaving observable gaps compared to manual splitting. Optimization opportunities also exist for the built-in schedulers, where more high-performance parallel primitives could be exploited to further reduce scheduling overhead. On the other hand, the placement of suspension points and the selection and configuration of schedulers still require human labor. It will be interesting to explore how automatic tuning schemes might be incorporated.

Another direction worth investigating is how to further enhance the expressiveness of the GPGPU programming model and how this expressiveness can be effectively utilized. Our GPU coroutine model serves as a good example of such exploration, and we believe that there are broader open areas in the GPU world for other constructs that can be transplanted from well-developed CPU languages.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (Project No. 62372257).

REFERENCES

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana) *(HPG '09)*. Association for Computing Machinery, New York, NY, USA, 145–149. https://doi.org/10.1145/1572769.1572792

Andrew W. Appel. 1992. *Compiling with continuations.* Cambridge University Press, USA.

Saman Ashkiani, Andrew Davidson, Ulrich Meyer, and John D. Owens. 2016. GPU multisplit. *SIGPLAN Not.* 51, 8, Article 12 (feb 2016), 13 pages. https://doi.org/10.1145/3016078.2851169

Sai Praveen Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Frédo Durand, Aaron Lefohn, and Yong He. 2023. SLANG.D: Fast, Modular and Differentiable Shader Programming. *ACM Trans. Graph.* 42, 6, Article 264 (dec 2023), 28 pages. https://doi.org/10.1145/3618353

Carlo Bergonzini. 2021. Lone Monk. https://blenderartists.org/t/lone-monk-cc0-scene-and-assets/1287621.

G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. 1979. *Simula Begin*. Chartwell-Bratt Ltd.

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

Blender Online Community. 2024. *Blender - A 3D Modelling and Rendering Package*. Blender Foundation, Stichting Blender Foundation, Amsterdam. http://www.blender.org

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (Los Angeles, California) *(SIGGRAPH '04)*. Association for Computing Machinery, New York, NY, USA, 777–786. https://doi.org/10.1145/1186562.1015800

G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. *SIGPLAN Not.* 17, 6 (jun 1982), 98–101. https://doi.org/10.1145/872726.806984

Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan. 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany) *(HWWS '02)*. Eurographics Association, Goslar, DEU, 69–78.

Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (jul 1963), 396–408. https://doi.org/10.1145/366663.366704

Oliver Danvy and Andrzex Filinski. 1992. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. 2014. Progressive Light Transport Simulation on the GPU: Survey and Improvements. *ACM Trans. Graph.* 33, 3, Article 29 (jun 2014), 19 pages. https://doi.org/10.1145/2602144

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

Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Chicago, IL, USA) *(Onward! 2021)*. Association for Computing Machinery, New York, NY, USA, 68–84. https://doi.org/10.1145/3486607.3486751

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. *SIGPLAN Not.* 28, 6 (jun 1993), 237–247. https://doi.org/10.1145/173262.155113

Tim Foley, Mike Houston, and Pat Hanrahan. 2004. Efficient partitioning of fragment shaders for multiple-output hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Grenoble, France) *(HWWS '04)*. Association for Computing Machinery, New York, NY, USA, 45–53. https://doi.org/10.1145/1058129.1058136

Kirill Garanzha and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum* 29, 2 (2010), 289–298. https://doi.org/10.1111/j.1467-8659.2009.01598.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01598.x

Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

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

Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (mar 2023), 81 pages. https://doi.org/10.1145/3570638

Ivor Horton and Peter Van Weert. 2023. *Class Templates*. Apress, Berkeley, CA, 669–716. https://doi.org/10.1007/978-1-4842-9343-0_17

Qiming Hou, Kun Zhou, and Baining Guo. 2008. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graph.* 27, 3 (aug 2008), 1–12. https://doi.org/10.1145/1360612.1360618

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

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

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

John Kessenich, Boaz Ouriel, and Raun Krisch. 2023. *SPIR-V Specification*. The Khronos Group Inc. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

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

Anselmo Lastra, Steven Molnar, Marc Olano, and Yulan Wang. 1995. Real-time programmable shading. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (Monterey, California, USA) *(I3D '95)*. Association for Computing Machinery, New York, NY, USA, 59–ff. https://doi.org/10.1145/199404.199414

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.

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

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

Chris D. Marlin. 1980. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Lecture Notes in Computer Science, Vol. 95. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-10256-6

David Mazières. 2021. *My tutorial and take on C++20 coroutines*. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html

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.

Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (feb 2009), 31 pages. https://doi.org/10.1145/1462166.1462167

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

Marc Olano and Anselmo Lastra. 1998. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/280814.280857

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4, Article 66 (jul 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

Amar Patel and Tex Riddell. 2024. *D3D12 Work Graphs*. https://devblogs.microsoft.com/directx/d3d12-work-graphs

Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. 2000. Interactive multi-pass programmable shading. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 425–432. https://doi.org/10.1145/344779.344976

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

Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/383259.383275

A. Rotenberg. 1960. A New Pseudo-Random Number Generator. *J. ACM* 7, 1 (jan 1960), 75–77. https://doi.org/10.1145/321008.321019

Matthew Rusch and Evan Hart. 2022. *Improve Shader Performance and In-Game Frame Rates with Shader Execution Reordering*. https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27, 3 (aug 2008), 1–15. https://doi.org/10.1145/1360612.1360617

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

```
int main(int argc, char *argv[]) {
  Context context{argv[0]};
  Device device = context.create_device("cuda");
  Stream stream = device.create_stream();
  Image<float> device_img = device.create_image<float>(
    PixelStorage::BYTE4, 512u, 512u);
  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_image_kernel = [&](ImageFloat image) {
    auto coord = dispatch_id().xy();
    auto size = make_float2(dispatch_size().xy());
    auto rg = make_float2(coord) / size;
    auto srgb = to_srgb(make_float3(rg, 1.f));
    image.write(coord, make_float4(srgb, 1.f));
  };
  auto fill = device.compile(fill_image_kernel);
  std::vector<std::byte> host_img(512u * 512u * 4u);
  stream << fill(device_img).dispatch(512u, 512u)
         << device_img.copy_to(host_img.data())
         << synchronize();
  save_image("color.png", host_img, 512u, 512u, 4u);
}
```

Listing 9. A usage example of Luisa.

202 (nov 2019), 15 pages. https://doi.org/10.1145/3355089.3356554
Gerald J. Sussman and Guy L. Steele. 1998. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation* 11 (1998), 405–439. https://api.semanticscholar.org/CorpusID:18040106
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
W. E. Thomson. 1958. A Modified Congruence Method of Generating Pseudo-random Numbers. *Comput. J.* 1, 2 (01 1958), 83–83. https://doi.org/10.1093/comjnl/1.2.83 arXiv:https://academic.oup.com/comjnl/article-pdf/1/2/83/1175362/010083.pdf
Xiaochun Tong, Hsueh-Ti Derek Liu, Yotam Gingold, and Alec Jacobson. 2023. Differentiable Heightfield Path Tracing with Accelerated Discontinuities. In *ACM SIGGRAPH 2023 Conference Proceedings* (Los Angeles, CA, USA) *(SIGGRAPH '23)*. Association for Computing Machinery, New York, NY, USA, Article 19, 9 pages. https://doi.org/10.1145/3588432.3591530
Guido Van Rossum et al. 2007. Python Programming Language.. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.
Niklaus Wirth. 1988. *Programming in Modula-2*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-83565-0
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 (nov 2022), 19 pages. https://doi.org/10.1145/3550454.3555463

A REVIEW OF THE LUISA SYSTEM

In this section, we briefly review the original Luisa system to provide readers with an overview of its architecture and capabilities. Luisa offers an embedded DSL in C++ for kernel programming, a runtime abstraction layer for resource management, and multiple optimized compute backends. Listing 9 shows a usage example of Luisa.

A.1 Embedded DSL for Kernel Programming

The DSL is embedded in pure C++, imitating native C++ syntax. The language constructs include

- *Types and variables.* The DSL variables are typed using wrapper templates such as `Var<T>` (e.g., `Var<uint2>`, alias `UInt2`).
- *Expressions and statements.* Arithmetic, logic, and assignment operators for DSL variables are overloaded to *record* operations rather than perform calculations.
- *Control flows.* Special macros prefixed with the `$` sign (e.g., `$if`, `$for`) are used to generate control-flow statements for recording purposes.
- *Kernels and callable functions.* Both can be constructed as C++ template classes from C++ functions or function objects, including lambda expressions.

Kernels written in the DSL are dynamically tracked and recorded into *abstract syntax trees* (ASTs) at runtime. The tracing of the syntax tree is based on the proxy objects of the `Var<T>` template class. Mathematical operators or function calls involving `Var<T>` do not compute results but instead add nodes to the ASTs.

After being traced, ASTs, along with captured runtime information, are later forwarded to backends for device code generation, compilation, and pipeline creation.

A.2 Unified Runtime Interfaces

Luisa provides a unified runtime abstraction layer across different compute backends. The interfaces support two major capabilities: (1) *resource management*, which supports common resources used by graphics APIs, including buffers, textures, bindless arrays, meshes, acceleration structures, shaders, streams, and events; and (2) *command encoding and submission* that abstracts GPU tasks such as data transferring and shader execution into commands, which are submitted via streams to the backends.

A.3 Dynamic Multi-Stage Programming

With the DSL embedded in the host C++ language and dynamically traced and compiled at runtime, developers have deep control over the backend shader code composition. This facilitates *multi-stage programming* [Taha 2004] with Luisa. Fig.14 shows an example of such a capability, where developers can write real loops with the DSL control-flow macros, programmatically unroll loops with native C++ control flows, or even dynamically compose device-side logic using a mix of both.

We exploit this ability to write generic coroutine schedulers. For example, we can instantiate state-machine schedulers with an arbitrary number of states that are only known at runtime.

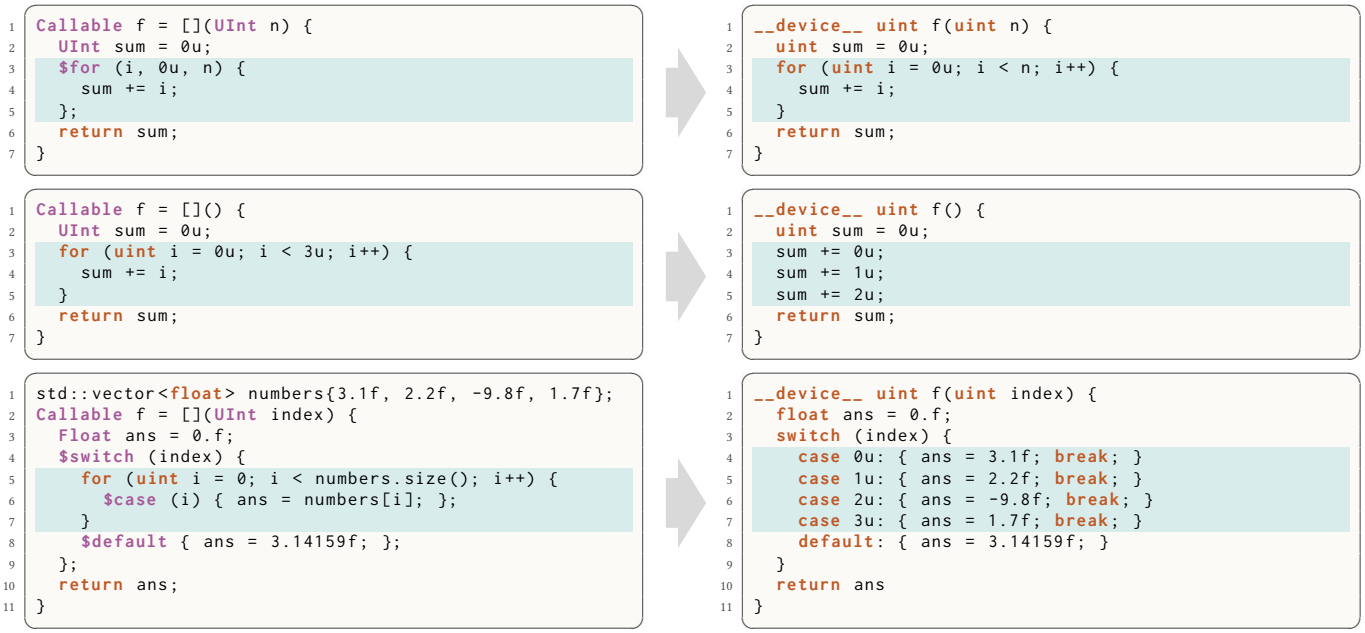
B IMPLEMENTATION OF THE HIGH-LEVEL PATTERNS

Many high-level asynchronous programming patterns and primitives can be effectively implemented using our relatively straightforward coroutine model. We take the aforementioned `Generator`s and `$await` features as examples and discuss their implementations.

B.1 Generators

A generator is indeed an object that yields a sequence of values to its caller lazily, producing each value only as needed upon each call. We can use the asynchronicity in coroutines to realize this laziness: the generator suspends itself after generating a value, and the caller resumes it only when more values are needed.

We follow this basic idea to implement the `Generator` class with our coroutine interfaces, shown in Listing 10. Note that since we



```
Callable f = [](UInt n) {
  UInt sum = 0u;
  $for (i, 0u, n) {
    sum += i;
  };
  return sum;
}
```

```
__device__ uint f(uint n) {
  uint sum = 0u;
  for (uint i = 0u; i < n; i++) {
    sum += i;
  }
  return sum;
}
```

```
Callable f = []() {
  UInt sum = 0u;
  for (uint i = 0u; i < 3u; i++) {
    sum += i;
  }
  return sum;
}
```

```
__device__ uint f() {
  uint sum = 0u;
  sum += 0u;
  sum += 1u;
  sum += 2u;
  return sum;
}
```

```
std::vector<float> numbers{3.1f, 2.2f, -9.8f, 1.7f};
Callable f = [](UInt index) {
  Float ans = 0.f;
  $switch (index) {
    for (uint i = 0; i < numbers.size(); i++) {
      $case (i) { ans = numbers[i]; };
    }
    $default { ans = 3.14159f; };
  };
  return ans;
}
```

```
__device__ uint f(uint index) {
  float ans = 0.f;
  switch (index) {
    case 0u: { ans = 3.1f; break; }
    case 1u: { ans = 2.2f; break; }
    case 2u: { ans = -9.8f; break; }
    case 3u: { ans = 1.7f; break; }
    default: { ans = 3.14159f; }
  }
  return ans
}
```

Fig. 14. An example of multi-stage programming with Luisa. The left column shows device functions written in Luisa's DSL, and the right column displays the (pseudo-)code of the generated backend shaders. Top: we can use the DSL control-flow macro `$for` to generate a real loop in the backend shader code. Middle: using the native C++ `for`-statement repeats the AST recording steps and generates unrolled device code. Bottom: mixed use of the native and DSL control flows allows developers to dynamically compose the kernel logic with host-side runtime information.

have restricted `Coroutines` to a void return type, the yielded value is passed through the coroutine frame via the `__yielded_value` field. The `$yield` DSL keyword is a macro that simply designates the yielded value and suspends the generator. Additionally, to accommodate the range-for syntax, the `begin()` and `end()` methods of `GeneratorIter` return wrapper objects that record an AST loop with a single-time pseudo-execution of the C++ `for` loop.

In graphics tasks, `Generator`s might be used to implement random number samplers, with the advantage of managing internal states without exposing them to the caller. Listing 11 provides an example of a uniform sampler based on a *linear congruential generator* (LCG) [Rotenberg 1960; Thomson 1958].

B.2 Chained Suspension with `await`

In many high-level CPU programming languages, a coroutine can use the `await` operator to suspend its execution until the awaited task (usually another coroutine) completes. This feature enables a logically chained suspension pattern for nested coroutine invocations, which is essential for managing complex asynchronous workflows in a structured manner.

Our GPU coroutine implementation also provides this capability with the `$await` keyword. Listing 12 demonstrates how this feature can aid in software encapsulation by allowing nested coroutine functions to manage internal suspension points.

Listing 13 provides the core reification for `$await`. A `Coroutine` returns a lambda upon invocation, representing a state machine that cycles through the subroutines and suspends after each step. The lambda's instructions are immediately recorded into the caller coroutine's AST, facilitated by the `$await` keyword. This effectively inlines the suspension operation into the caller, recursively suspending it at points where the nested coroutine would suspend.

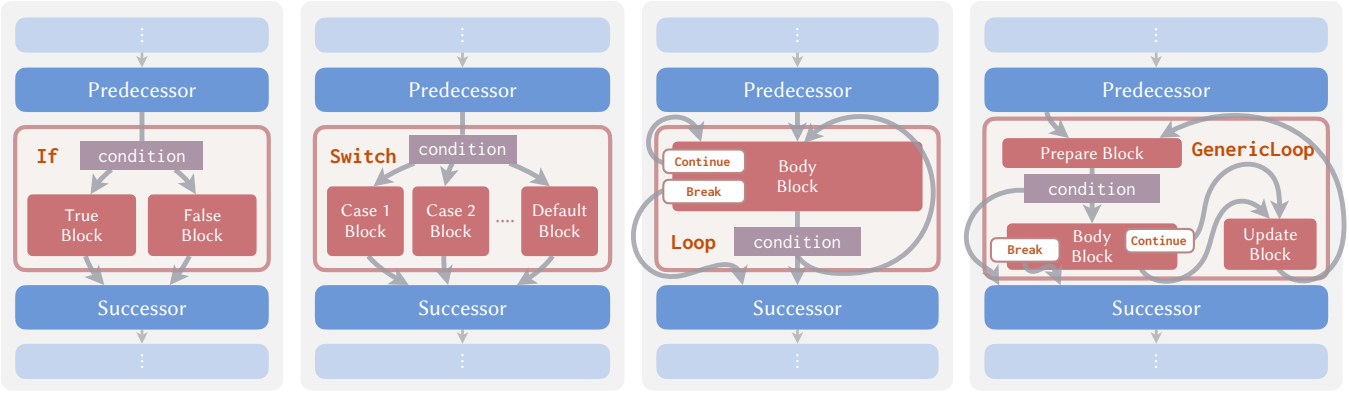
C COMPILER ANALYSIS AND TRANSFORMATION

C.1 Preprocessing

Luisa's C++-embedded DSL uses ASTs as the unified representation for device functions. However, the recursive tree structures are less suitable for analysis and manipulation. For this reason, we first translate the ASTs to a streamlined IR to facilitate the coroutine transformation and analysis passes.

The IR was originally contributed by Tong et al. [2023] for automatic differentiation with Luisa and has been merged into the main code repository. It is in the *static-single-assignment* (SSA) form with *structured* control-flow instructions (including `If`, `Switch`, `Loop`, and `GenericLoop`, as listed in Fig. 15). Two less SSA-styled and structured constructs, however, are (1) the `Local` instruction that declares thread local storage for variables, similar to `alloca` in LLVM [Lattner and Adve 2004] and `OpVariable` in SPIR-V [Kessenich et al. 2023]; and (2) permission for `Break`, `Continue`, and early `Return` instructions in the structured control flow. Both exist to relieve the translation burdens from the frontend ASTs.

C.1.1 Control flow normalization. The control-flow instructions, `GenericLoop`, `Break`, `Continue`, and early `Return`, ease frontend programming but also complicate the programs' control flow. To simplify the subsequent passes, we eliminate them in the control flow normalization pass.



(a) (b) (c) (d)

Fig. 15. Control-flow instructions in Luisa's IR, including (a) `If`, (b) `Switch`, and (c) `Loop`, which resemble the `if`, `switch`, and `do-while` statements in C++, respectively; (d) `GenericLoop` has a relatively more complex semantic, shown as Listing (a) in Fig. 16, which is designed to ease the frontend support of `for` loops. `Break` and `Continue` instructions, as well as early `Return`s, are allowed in the loops.

We can rewrite a `GenericLoop` using `do-while` loops and `if` statements with an auxiliary flag tracking whether to leave the loop, as shown in Fig. 16. This effectively lowers `GenericLoop`s to the `Loop` and `If` instructions. For `Break` and `Continue` removal, we adopt an approach similar to SLANG.D [Bangaru et al. 2023]. The basic idea is to use auxiliary flags to track if we should break or continue the loop during each iteration, and mask the successor instructions accordingly until we reach the loop exit. Unlike SLANG.D, however, we maintain an instruction stack to recursively mask successor instructions in each parent block until the function entry block, so that we can process early `Return`s embedded in arbitrarily nested loops and branches. An example is shown in Fig. 17. Basic constant folding of the loop conditions is also performed on the fly and trivially redundant loops, like `do {...} while (false)`, are flattened into their parent blocks if detected.

After this transformation pass, `If`, `Switch`, and `Loop` are the only remaining control-flow instructions in the program. Effectively, this ensures a *reducible* control flow graph (CFG) with back edges only inside the `Loop` instructions.

C.1.2 Live state reduction. It is important to Keep coroutine state frames lean and mean by reducing the live states at each suspension point. Otherwise, the large number of concurrent threads will magnify any small waste of per-thread memory access. We employ a set of passes for live state reduction, among which we find two passes very effective:

(1) *Variable scope decision* that attempts to shrink the live interval of a `Local` variable by demoting its declaration site to the innermost possible basic block; and
(2) *Load chain deferring* that attempts to reduce the used states by replacing the element extraction of a loaded aggregate with the loading of the pointer to that element (i.e., from `ExtractElem(Load(v), i)` to `Load(GetElemPtr(v, i))`).

For example, in the path tracing task (Sec. 7.1) on the *Lone Monk* scene, the coroutine state frame size decreases from 736 bytes to 224 bytes with the variable scope decision pass and further to 208 bytes with load chain deferring.

C.2 Continuation Extraction

The continuation at a suspension point is the instructions to be executed when resumed. Therefore, the general task is to identify and collect all instructions reachable from each suspension point. The control flow in the continuation should also be reconstructed so the transformed program produces identical results to the original.

Note that for compatibility with the structured IR design, the reconstructed control flow must stay structured, using only the instructions listed in Fig. 15; arbitrary jumps like `goto` are disallowed.

C.2.1 CFG distillation. The continuation extraction process reorganizes a coroutine into per-suspension-point code scopes, during which instruction motion, duplication, deletion, etc., may occur. To avoid destructive modification on the original IR and ignore the unrelated details about non-control instructions, we distill the program into a control flow graph (CFG).

A graph may contain multiple code scopes, each representing the entry or an individual continuation. The IR instructions in the original program are wrapped into *graph nodes*, constituting the scopes. Three node categories are present:

(1) *Simple nodes* that *reference* (rather than duplicate) the non-control instructions in the original IR;
(2) *Control-flow nodes*, including `If`, `Switch`, and `Loop`, which replicate the program's logical structure; and
(3) *Terminator nodes*, `Suspend` and `Terminate`, which end the current scope and transfer the control back to the scheduler.

Initially, the graph is a trivial dump of the original program, with a single, preliminary scope of nodes wrapping all IR instructions in the coroutine body. Successive stages then collect the reachable nodes into individual continuation scopes and reconstruct the control flow at each suspension point.

```
Prepare:
    /* ... */
    goto (condition ? Body : Exit);
Body:
    /* ... */
    continue => { goto Update; }
    break    => { goto Exit; }
    /* ... */
    goto Update;
Update:
    /* ... */
    goto Prepare;
Exit:
```

(a) The semantic of the `GenericLoop` instruction

```
do {
    [[ instructions in the Prepare block ]]
    if (!condition) break;
    bool should_leave = false;
    do {
        [[ instructions in the Body block, where
           continue => { break; }
           break    => { should_leave = true; break; }
        ]]
    } while (false);
    if (should_leave) break;
    [[ instructions in the Update block ]]
} while (true);
```

(b) The lowered `GenericLoop` using `if` and `do-while` with an auxiliary flag

Fig. 16. Code template that lowers the `GenericLoop`. We rewrite a `GenericLoop` using the `if` statements and `do-while` loops (i.e., the `If` and `Loop` instructions in the IR). An auxiliary flag tracking is created to track whether we should leave the loop.

```
do {
  /* ... */
  if (c1) {
    return; // early return
  }
  /* ... */
} while (c2);
/* ... */
return;
```

```
bool early_returned = false;// auxiliary tracking flag
do {
  /* ... */
  if (c1) {
    // the original early return is tracked by the flag
    early_returned = true;
  }
  // mask the successor instructions
  if (!early_returned) {
    /* ... */
  }
} while (c2 && !early_returned);// changed condition
// mask the successors in parent blocks recursively
if (!early_returned) {
  /* ... */
}
// until the function-level return
return;
```

Fig. 17. An example of the early `Return` removal pass.

C.2.2 Reachable node collection. The CFG is always reducible with the only back edges inside the `Loop` nodes, as ensured by the control flow normalization pass (Sec. C.1.1). This property leads to a series of simple rules in reachable node identification. For any node p in the CFG, let $\mathcal{R}(p)$ denote the reachable node set from p, we have

(1) If p is a `Loop` node, then $p \in \mathcal{R}(p)$;
(2) If q is the successor of p, then $q \in \mathcal{R}(p)$;
(3) If $q \in \mathcal{R}(p)$, then $\mathcal{R}(q) \subseteq \mathcal{R}(p)$; and
(4) If q is a control-flow node (i.e., `If`, `Switch`, or `Loop`), let $\mathcal{B}(q)$ denote a nested basic block inside it, then
    - $\forall s \in \mathcal{B}(q), \mathcal{R}(q) \subseteq \mathcal{R}(s)$; and
    - $\forall s \in \mathcal{B}(q), q \in \mathcal{R}(p) \Rightarrow s \in \mathcal{R}(p)$.

Practically, these rules can be implemented as a depth-first traversal over the CFG from the entry. We maintain a stack of the parent control-flow nodes along the traversal path. When a `Suspend` node is encountered, we backtrack the node stack and collect the reachable nodes recursively into a newly created code scope (Fig. 18). Some on-the-fly simplification strategies are also considered. For example, nodes dominated by another `Suspend` node are not considered reachable from the current suspension point.

C.2.3 Control flow reconstruction. The program logic must be recovered from the collected reachable nodes, so the resumed coroutine produces the same results as if never suspended.

For suspension points not contained in a `Loop`, the reconstruction is straightforward by simply joining the reachable nodes in the depth-first order. Otherwise, back edges may require the inclusion of predecessor nodes visited before. A direct idea might be unrolling all successor nodes before the re-entrance into the outermost loop. However, this possible solution may duplicate an unpredictable amount of unrolled code when loops are deeply nested.

Instead, we duplicate the relevant control-flow nodes once and replay the condition values that lead to the current suspension point at the first run after resumption. Also, during the first run, the nodes preceding the suspension point should be skipped, so a new control-flow node, `SkipOnReplay`, is added to each parent block to contain these preceding nodes. Fig. 19 shows an example of this transformation strategy.

After this stage, the CFG is populated with appropriately structured subroutine scopes, each corresponding to the continuation of a suspension point; plus a special one for the entry.

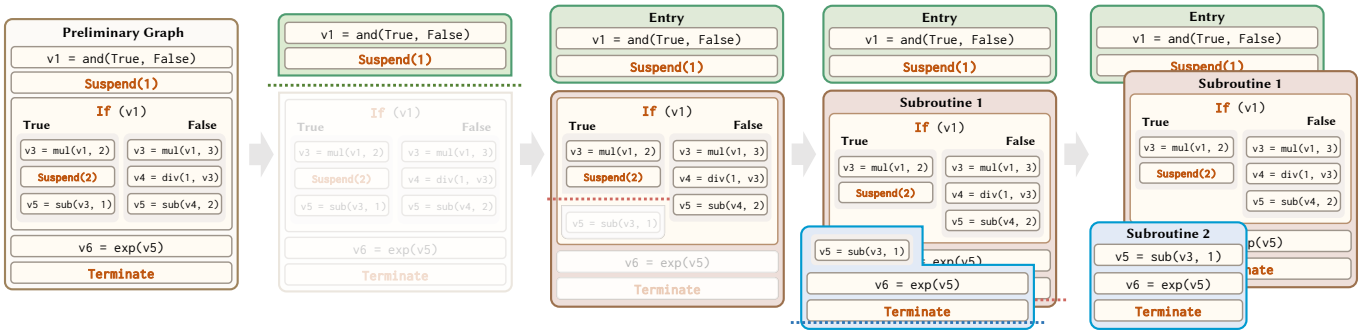


Fig. 18. Recursive splitting of the coroutine control flow graph into subroutine scopes of reachable nodes.

```cpp
template<typename T, typename... Args>
struct Generator<T(Args...)> {
  Coroutine<Args...> coro;
  GeneratorIter<T> operator()(Args... args) const {
    CoroFrame frame = coro.instantiate();
    coro.entry()(frame, args...);
    return GeneratorIter<T>{
      .n = coro.subroutine_count(),
      .frame = std::move(frame),
      .resume = [=, this](CoroFrame &f, CoroToken i) {
        coro[i](f, args...);
      },
    };
  }
};

template<typename T>
struct GeneratorIter {
  uint n;
  CoroFrame frame;
  function<void(CoroFrame &, CoroToken)> resume;

  Bool is_terminated() const {
    return frame.is_terminated();
  }

  void update() const {
    // state machine to select the next subroutine
    $switch (frame.target_token) {
      // note: entry (token 0) is included as well
      for (uint i = 0; i < n; i++) {
        $case (i) { resume(frame, i); };
      }
    };
  }

  T value() {
    return frame.get<T>("__yielded_value");
  }

  // wrappers to support the range-for syntax
  GeneratorRangeForIter begin() {/*...*/}
  GeneratorRangeForIter end()   {/*...*/}
};

#define $yield(x)                          \
  do { $promise("__yielded_value", x); \
       $suspend();                         \
  } while (0)
```

Listing 10. Implementation of the `Generator` class based on our coroutine interfaces. The asynchronicity of coroutines facilitates the lazy evaluation of the yielded values. Listing 1 demonstrates its usage.

```cpp
Generator<float(uint)> lcg_sampler = [](UInt seed) {
  UInt state = seed;
  $loop { // infinite loop: as many numbers as desired!
    state = 1664525u * state + 1013904223u;
    Float u = state * 0x1p-32f; // map to [0, 1)
    $yield(u);
  };
};

// usage
auto rng = lcg_sampler(2024u);
Float a = rng.update().value();
Float b = rng.update().value();
...
```

Listing 11. Example implementation of a uniform sampler based on a linear congruential generator (LCG), using the built-in `Generator` feature.

```cpp
// nested coroutine for ray intersection
Coroutine trace = [&](Var<Ray> ray, Var<Hit> &hit) {
  $suspend();
  hit = scene->intersect(ray);
};
// nested coroutine for surface shading
Coroutine shade = [&](Var<Hit> hit, Float3 &Li) {
  $if (hit->miss()) {
    Li = make_float3(0.f);
  } $else {
    $suspend();
    Li = scene->sample_direct_lighting(hit);
  };
};
// main coroutine that implements the rendering logic
Coroutine render = [&](...) {
  Var<Ray> ray = camera->generate_ray(pixel_id);
  Var<Hit> hit;
  $await trace(ray, hit);
  Float3 Li;
  $await shade(hit, Li);
  film->accumulate(pixel_id, Li);
};
```

Listing 12. Example usage of the `$await` keyword for software encapsulation, where nested coroutine functions designate internal suspension points for ray intersection and surface shading.

C.3 State Frame Materialization

When resuming a coroutine, the program context, i.e., the data and states that the continuation requires, should also be passed on. Nevertheless, it is impractical to simply dump all the variables

```
template<typename... Args>
auto Coroutine<Args...>::operator()(Args... a) const {
  return [=, this]() {
    CoroFrame frame = this->instantiate();
    this->entry()(frame, a...);
    $while (!frame.is_terminated()) {
      $suspend(); // this will suspend the caller
      $switch (frame.target_token) {
        uint n = this->subroutine_count();
        for (uint i = 1; i < n; i++) {
          $case (i) { (*this)[i](frame, a...); };
        }
      };
    };
  };
}

// helper class for the $await syntactic sugar
struct CoroutineInvocationAwaiter {
  template<typename F>
  void operator%(F &&f) const { f(); }
};

#define $await \
  CoroutineInvocationAwaiter{} %
```

Listing 13. Implementation of the `$await` keyword based on our coroutine interfaces. This feature enables a logically chained suspension pattern for nested coroutine invocations, as demonstrated in Listing 2.

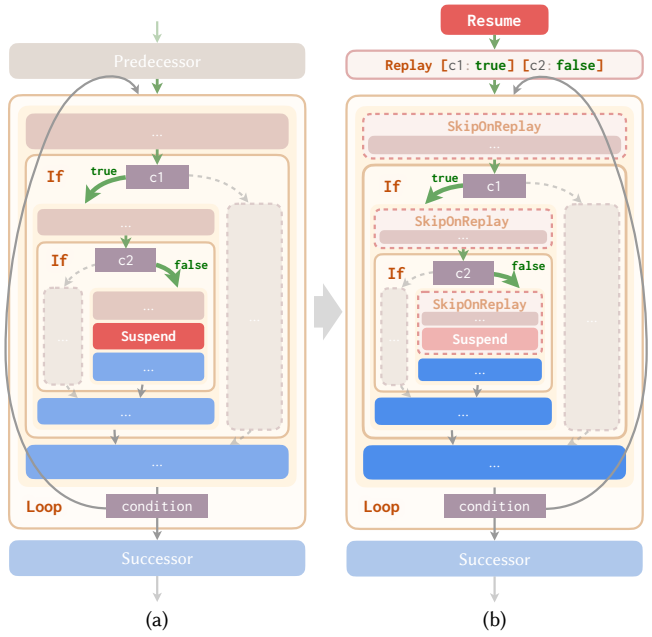


Fig. 19. Control flow reconstruction with condition value replay. To resume the program logic at the suspension point (a), we duplicate the whole related nodes and replay the condition values (b) that have led the control flow (green arrows) to the suspension point. Preceding nodes to the original suspension point are wrapped into a special `SkipOnReplay` node at the first loop entrance after the resumption.

and then restore them on resumption. Memory traffic and register pressure might soon become the performance bottleneck.

The ideal approach is to only load and store the necessary subset of the live states. In our implementation, we honor the rules below:

(1) If a value used in the current subroutine might reference external definitions in other subroutines, then it should be loaded on resumption; and
(2) If a value defined in the current subroutine might be loaded by other subroutines, then it should be stored on suspension.

We perform data-flow analysis on the CFG to compute these sets: first with an intra-scope *use-define analysis* on each subroutine to find the internal definitions and external references; then with an inter-scope *liveness analysis* on the transition graph to determine the values to store and load on each suspension-resumption edge.

Notably, for aggregates (i.e., vectors, matrices, arrays, and structures), the analysis is done at the field level as they are decomposed into scalar elements. This helps reduce the frame size by discarding inactive fields (Fig. 21).

C.3.1 Intra-scope use-define analysis. While the standard SSA form implicitly conveys the use-define information, our extension of the `Local` instruction allows updating a variable multiple times and thus requires extra explicit handling.

A forward data-flow analysis is leveraged for this purpose. For each subroutine scope s in the CFG, we traverse its nodes to track three cumulative sets of variables when *reaching* a node p (before it has been processed and stepped over):

(1) $\mathcal{K}(p)$ for variables whose definitions are confirmed *killed* in the current scope, i.e., their values are always overwritten on all code paths since the resumption to p;
(2) $\mathcal{E}(p)$ for variables that might reference *external* definitions in other scopes, i.e., their values are not internally killed when reaching some of their users before p; and
(3) $\mathcal{T}(p)$ for variables that are possibly *touched* (i.e., values modified) in the current scope before p.

All sets are initialized empty. The following rules are then applied at each node p during the traversal, to update the sets when reaching p's successor node q:

- If p is a simple node that locally uses variables of $\mathcal{U}(p)$, defines $\mathcal{D}(p)$, and modifies $\mathcal{M}(p)$, then

$$\begin{aligned}\mathcal{E}(q) &= \mathcal{U}(p) - \mathcal{K}(p),\\ \mathcal{K}(q) &= \mathcal{D}(p) \cup \mathcal{K}(p),\\ \mathcal{T}(q) &= \mathcal{M}(p) \cup \mathcal{K}(p).\end{aligned}$$

- If p is an `If` or `Switch`, we process each child block $\mathcal{B}_i$ independently with $\mathcal{K}(p)$, $\mathcal{E}(p)$, and $\mathcal{T}(p)$ inherited as its input, and then merge the block-wise results as

$$\begin{aligned}\mathcal{E}(q) &= \bigcup_i \mathcal{E}(\mathcal{B}_i),\\ \mathcal{K}(q) &= \bigcap_i \mathcal{K}(\mathcal{B}_i),\\ \mathcal{T}(q) &= \bigcup_i \mathcal{T}(\mathcal{B}_i).\end{aligned}$$

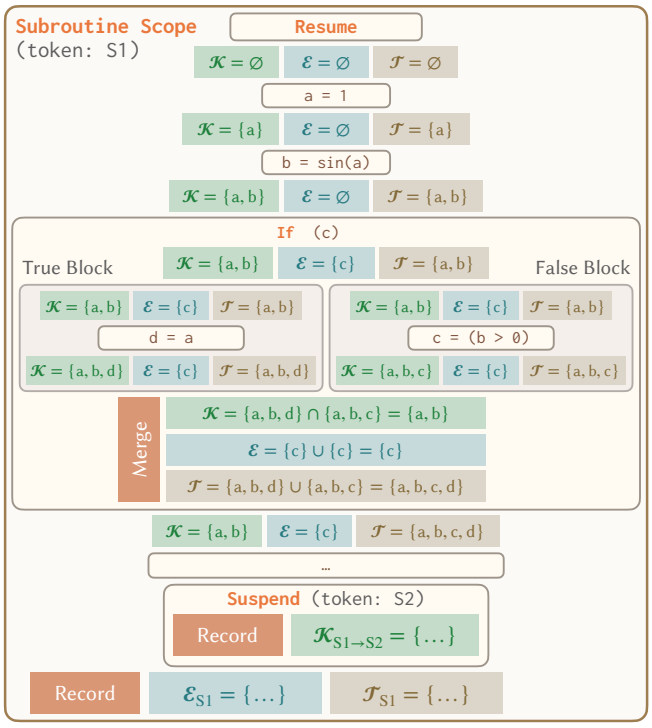


Fig. 20. An example of the intra-scope use-define analysis process.

- If p is a `Loop`, we can simply process the nodes in its body as if they are inlined into the parent block.
- If p is a `Suspend`, we record $\mathcal{K}(p)$ with the suspension token t as $\mathcal{K}_{s \to t}$ for later inter-scope analysis (Sec. C.3.2).
- The `Terminate` node may only appear at the last of the scope, thus requiring no special handling.

After this analysis, per-scope sets of the external references $\mathcal{E}_s$ and the internally touched variables $\mathcal{T}_s$ are computed for each scope s; as well as the per-suspension sets of the killed variables $\mathcal{K}_{s \to t}$ for each reachable suspension token t inside each scope s. Fig. 20 gives an example of the data-flow analysis process.

C.3.2 Inter-scope liveness analysis. The union set of external references in all scopes constitutes the coroutine state frame. However, not all states are necessarily loaded on every resumption or stored on every suspension: a subroutine might access only a portion of it.

To determine the demanded load/store subsets, we first construct a coroutine transition graph for the subroutine scopes. As described in Sec. 5.2, each graph node corresponds to a subroutine and the directed edges between nodes represent the suspension-resumption relations (Fig. 6).

Liveness analysis is then performed on the graph to identify the live states at the beginning of each scope, i.e., the variables that are needed by this scope or any reachable scope from it. Let $\mathcal{L}_s$ denote the live state set at the beginning of a scope s, it is the solution to the following data-flow equation:

$$\mathcal{L}_s = \mathcal{E}_s \cup \bigcup_{s \to t} (\mathcal{L}_t - \mathcal{K}_{s \to t}) .$$

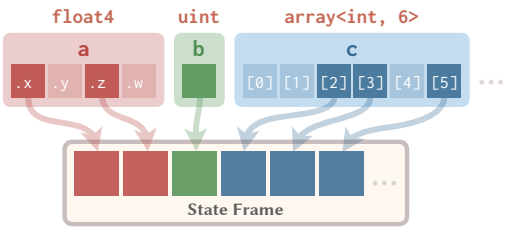


Fig. 21. Layout compaction of the coroutine state frame. We employ field-level data-flow analysis on aggregate variables to detect their live elements. Inactive elements are excluded from the resulting frame to reduce its size.

We start with $\mathcal{L}_s = \varnothing$ for each scope s and iterate over the transition graph until the fixed point to solve this equation.

Based on the liveness information, we can now easily compute the input set

$$\mathcal{I}_s = \mathcal{E}_s \cup \bigcup_{s \to t} ((\mathcal{L}_t - \mathcal{K}_{s \to t}) \cap \mathcal{T}_s)$$

for a scope s to load on resumption and the output set

$$O_{s \to t} = \mathcal{L}_t \cap \mathcal{T}_s$$

for s to store on each suspension with t as the resumption target.

Intuitively, $\mathcal{I}_s$ includes states that are either (1) directly required by the current scope s or (2) needed by adjacent scopes but unfortunately touched but not killed by s; and in $O_{s \to t}$ are the live states that are needed by the target scope t but possibly modified in the current scope s.

C.3.3 Frame structure layout. We then arrange the coroutine states into a frame. In the current implementation, we simply flatten and tile the fields of each state in the state frame structure. As mentioned before, since the data-flow analysis passes are performed at the field level for aggregates, we can compact the frame layout by excluding the uninvolved aggregate members (Fig. 21). This may help reduce the memory traffic in state frame management. A future implementation may further condense the state frames via register allocation algorithms such as *graph coloring* [Chaitin 1982].

C.4 Postprocessing

Now that we have a full knowledge of the control transfer and state transition of the coroutine, we can instantiate it into concrete `Callable` functions.

We first reconstruct the IR from the outlined and transformed subroutine scopes in the CFG. Coroutine-specific intrinsic nodes are lowered and rewritten as normal IR instructions. For example, `Suspend` is converted to an update of the target token field in the state frame followed by a `Return` instruction. Then the IR is translated back to `Callable` ASTs so the frontend may conveniently reorganize the subroutines into other forms.

The transition graph is fed back to the frontend as well, which can be later leveraged for runtime optimization, as done in the built-in schedulers.