High-Performance Graphics (2025)
A. Knoll and C. Peters (Editors)

No More Shading Languages: Compiling C++ to Vulkan Shaders

H. Devillers[1], M. Kurtenacker[3], R. Membarth[2,3], S. Lemme[3], M. Kenzel[3], Ö. Yazici[1], and P. Slusallek[1,3]

[1]Saarland University (UdS), Saarland Informatics Campus, Germany
[2]Technische Hochschule Ingolstadt (THI), Research Institute AImotion Bavaria, Germany
[3]Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarland Informatics Campus, Germany

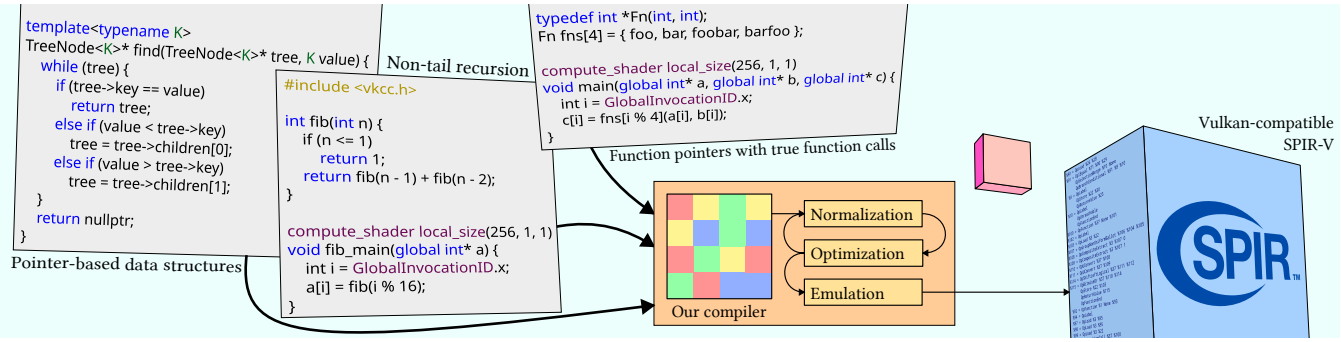

Figure 1: *Our compiler offers support for a variety of language features usually unavailable in graphical shaders.*

Abstract

Graphics APIs have traditionally relied on shading languages, however, these languages have a number of fundamental defects and limitations. By contrast, GPU compute platforms offer powerful, feature-rich languages suitable for heterogeneous compute. We propose reframing shading languages as embedded domain-specific languages, layered on top of a more general language like C++, doing away with traditional limitations on pointers, functions, and recursion, to the benefit of programmability. This represents a significant compilation challenge because the limitations of shaders are reflected in their lower-level representations. We present the Vcc compiler, which allows conventional C and C++ code to run as Vulkan shaders. Our compiler is complemented by a simple shading library and exposes GPU particulars as intrinsics and annotations.
We evaluate the performance of our compiler using a selection of benchmarks, including a real-time path tracer, achieving competitive performance compared to their native CUDA counterparts.

CCS Concepts
• ***Software and its engineering*** → ***Compilers;*** • ***Computing methodologies*** → *Rendering; Ray tracing;*

1. Introduction

Even though both real-time graphics and compute APIs run on the same hardware, and share many concepts and functionalities, the finer points of their programming models are surprisingly divergent.

The evolution of programming models for graphics has tracked the hardware's, which has evolved from fully fixed-function 3D accelerators to increasingly programmable and general-purpose SIMD machines. Because of this smooth transition, happening over a long time, and the practical demands of backwards compatibility, a lot of vestigial features and designs persist in contemporary graphics APIs [Sam17].

The main interest of this paper, shading languages, are domain-specific languages with C-like syntax, used to define the programmable stages of graphics pipelines. Most shading languages still in use today like GLSL [Kes05] and HLSL [PM03] were created when the scope of shader programs was still small, leading to them forgoing many features such as pointers, recursion, or virtual functions that could not be effectively supported. These feature omissions have mostly stood, meanwhile the scope of shader programs has not stayed small, leading to considerable pains scaling up programs.

GPU compute APIs have taken a shorter, more straightforward path to where they are now. They offer a generic SIMT program-

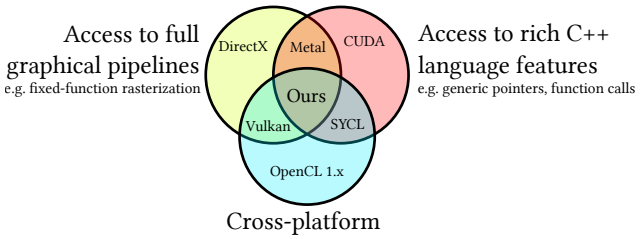

Figure 2: *Diagram of the compromises that must be made when deciding on a GPU programming interface.*

ming model, operating on abstract work-items forming a 3D grid. As the application domain is not known, programming models allowed the user to define arbitrary complex data structures and control flow from the start, including pointer support. There are few domain-specific features such as those found in graphical APIs, in particular texture support is very limited and the conventional rendering pipeline is not accessible. This is a practical issue for writing real-time graphics applications, such as games, because it means ignoring the hardware acceleration that graphics APIs leverage, which is a tough proposition for achieving ideal performance.

As discussed in more detail in Section 2, all mainstream programming models fall pretty decidedly in one category or the other. Due to this fractured software landscape, application developers must make compromises between portability, access to hardware features and language expressivity. When targeting the GPU, there is no solution that provides the best of all worlds. Figure 2 visually represents the current state of the ecosystem and shows where our solution would fit.

Graphics APIs do support so-called "compute shaders" which work like the dedicated compute grid models, but use conventional shading languages. This is a significant obstacle for the reasons we have outlined earlier. If we could use a graphics API with a substantially improved language to write shaders in, ideally a C++ dialect comparable to those used in GPU Compute, there would be a viable path forward for a unified compute and graphics API.

One of the motivations for this work is to study the feasibility of making Vulkan into that "best of both worlds" API, tackling the shading language issues by replacing them outright with C++, although our approach is applicable to other languages such as Rust.

The overall thesis of this work is that current shading languages are needlessly restrictive when considering modern hardware, and that an approach based on extending a general-purpose language with domain-specific constructs is conceptually simpler, no less effective and opens new exciting possibilities.

In summary, this paper makes the following contributions:

- We offer an overview of the current landscape of GPU programming languages in Section 2 and related work in Section 3.
- In Section 4 we elaborate on the practical implication of using C++ as embedded (rather than external) shading language.
- Compiling C++ shaders to the limited environment of Vulkan SPIR-V shaders comes with a number of technical challenges, tackled by our compiler. We study those in Section 5.
- Our prototype implementation successfully demonstrates the overall concept, and present various applications in Section 6, including reference implementations in conventional technologies.

2. Background

CUDA [Kir*07] is a well-established platform for GPU computing. CUDA offers a "single-source" programming model: functions can be compiled for the CPU, GPU, or both, depending on annotations. This avoids duplicating shared logic and makes porting existing applications to the GPU easier. To further ease the burden on programmers porting their code to the GPU, CUDA supports most of C and C++ language features, including full pointer support, function pointers, and recursion. `goto` statements are also supported.

OpenCL and SYCL are cross-platform standards focused on programming parallel processors. OpenCL [Mun09] uses a non-unified programming model: GPU kernels are written in a specific dialect of C or C++, and are compiled separately from host code. Unlike in CUDA, recursion and function pointers are not supported. SYCL [RL16] is a C++ library that provides abstractions for writing single-source heterogeneous programs in C++, comparable to what CUDA offers. While SYCL was initially explicitly tied to OpenCL, this requirement has since been relaxed to facilitate SYCL implementations on other platforms.

GLSL and HLSL Graphics APIs such as OpenGL and Vulkan feature a partially programmable graphics pipeline, with certain stages implemented by user-defined programs called shaders. Shader programs are traditionally written in domain-specific languages, such as OpenGL Shading Language (GLSL) [Kes05; RLG*09] or High Level Shading Language (HLSL) [PM03]. These languages are syntactically similar to C [Sam17], but do not offer support for pointers, union types, or recursion. Due to these restrictions, it is less practical to port existing code to shading languages, and the ability to share code between the CPU and GPU is hampered.

Metal [App24] is the proprietary API used on Apple devices. It is similar to APIs such as Vulkan but of note is the fact it uses a C++ dialect for shaders, Metal Shading Language (MSL). Of interest in this paper is the fact MSL supports pointers and especially function pointers in shaders, the only documented platform to do so. Metal also supports recursion, but requires the programmer to specify the maximal recursion depth. We did not evaluate Metal in detail in this work, but we note it tackles many of the concepts discussed in this paper.

SPIR-V [KOK18] is a standardized program representation used in graphics and compute APIs, including OpenCL and Vulkan. SPIR-V as a language is similar to LLVM IR, but natively supports graphics-relevant concepts such as vector and matrix operations, GPU resource descriptors, and passing data across pipeline stages. While SPIR-V can be consumed by both compute and graphics APIs, many instructions and features are optional (called *capabilities*). These capabilities are arranged in such a way that there are two significant dialects: a "kernel" and a "shader" dialect, corresponding to OpenCL and Vulkan respectively. Despite being a different

kind of pipeline, Vulkan compute shaders still use the "shader" dialect: Vulkan does not support "kernel" programs at all. Targeting this dialect presents a major challenge, as it inherits many of the limitations of conventional shading languages. In this work we are interested in the "shader" dialect since it is the one Vulkan supports.

3. Related Work

3.1. Languages Targeting Shader Execution

LARSEN's master thesis [Lar19] implemented a Vulkan backend for Futhark as an alternative to OpenCL. Rust GPU [Rus] is a promising project that aims to allow compilation of Rust directly to SPIR-V.

Slang [HFF18] is a new shading language that targets Vulkan. Of note, Slang features an *interface* abstraction that can fill the role of function pointers in certain scenarios. These interfaces are implemented by either specialization or generating a large monolithic shader. Implementation via actual function pointers is also possible, but only available on CUDA targets.

3.2. Program Translation to SPIR-V

A number of OpenCL approaches like Clspv [Goo] and Rusticl [Kar] as well as SYCL implementations such as Sylkan [TGF21] are being built on top of Vulkan. They face many of the same challenges tackled in this work, such as structured control flow and logical addressing. However, they do not present solutions for recursion and function pointers, as the source API does not support them either. The Khronos Group provides a bidirectional LLVM and SPIR-V translator [Thea], however, it only works with `Kernel` programs used by OpenCL. MOLL's bachelor thesis [Mol11] tackled transpiling a subset of LLVM IR [LLV] to GLSL which shares many structural constraints with the "shader" of SPIR-V.

4. Using C++ as a Shading Language

Shading languages are domain-specific languages, which means they facilitate programming in a particular domain, here real-time graphics. Traditional languages are external shading languages, meaning they are constructed as stand-alone languages: they have bespoke grammar, syntax and type systems, and are implemented in specialized compilers. Embedded domain-specific languages by contrast are implemented by co-opting the syntax and constructs of a host language and can get implemented as a specialized extension to existing compilers, or even entirely as libraries [LBH*18]. We found two compelling reasons why shading languages in particular are not a good fit to be external domain-specific languages:

- It is very much possible, if inconvenient, to write general-purpose code in shading languages. Shading languages were originally envisioned as, and stay similar to, C++ derivatives. This is reflected in their syntax, type systems, use of the C pre-processor and similar semantics. The domain-specific aspects of shading languages are additive to their C++ subset roots.
- The converse is also true: it is possible to write graphics code in C++. Language features such as overloading and templates make it practical to implement alternatives to the domain-specific functionality found in shading languages. This idea goes hand in hand with the premise of this work, and we put this idea to practice in Section 4.1.

Additionally, there is an argument to be made about the burden of maintaining shading languages. Because compute APIs do not need them, their use-case is extremely narrow, not only are they only usable on the GPU, they are only useful for graphics tasks. As they are not formally defined as C++ variants, they generally do not benefit from large consolidated efforts such as LLVM.

4.1. The Shading Language As a Library

We built the Not A Shading Language (NASL), a small header-only library that contains all the relevant fundamentals of functionality typically offered as language features in conventional shading languages. This library implements a sizable subset of the built-in types and functions found in GLSL, and is to an extent compatible with it. We implemented the following features, on a per-need basis for the applications we built so far:

- Small vectors (`vec4`, `vec3`, `uvec2`, ...) are one of the most recognizable features of shading languages, but are almost trivial to implement as C++ templated classes. The interesting challenge was getting swizzling functionality to work: we opted for a combination of macros and smart union members to offer arbitrary swizzles for up to 4 components.
- Small matrices (`mat4`), including transposition and vector multiplication.
- Rendering specific utility functions such as normalization, cross and dot product.

Clang does feature native support for swizzles in OpenCL mode, which we were not able to take advantage of since we used the standard C++ front-end. More invasive modifications of Clang were avoided to keep the scope of our work down, but would potentially be beneficial for a more mature implementation.

4.2. C++ Language Extensions

Some features found in shading languages are not just built for convenience but expose actual API and in turn hardware features. Examples include I/O variables to consume and produce pipelined data, texture handles, accesses and sampling operations, as well as buffers with special purposes, such as uniform buffers.

The Clang compiler allows us to annotate declarations with arbitrary strings, which we co-opt to specify custom annotations such as tagging functions as entry points, specifying attributes such as workgroup size and declaring intrinsic functions. Clang also exposes LLVM native vector types, which we use in place of NASL vectors where we have specific interface requirements, such as in shader inputs and outputs.

We provide a convenience header file with macros for all these annotations.

4.3. Benefits of C++ for Shaders

Neither NASL nor our language extensions are particularly advanced, and simply act to fill what would otherwise be a gap in

```
int f(int* ptr) {
    return *(ptr + 1);
}

float g(char* ptr) {
    return *(float*)ptr;
}

int h(void** ptr) {
    int i = 0;
    while (ptr) {
        i++;
        ptr = (void**) *ptr;
    }
    return i;
}
```

Listing 1: *Examples of problematic pointer usage: `f` performs unsafe pointer arithmetic, `g` reinterprets the pointer and `h` loads a pointer from memory. Note how these functions cannot be written with C++ references either.*

functionality. This confirms what we suggested at the start of this section: shading languages just do not have a lot of shading functionality, and what they have does not require them to be separate languages.

Instead the main benefit of our approach are that code sharing is made not just possible but practical, and that shaders can use the richer featuresets of the host language. In our case, this includes unrestricted data pointers, generic addressing, and function pointers.

5. Overcoming the Limitations of SPIR-V Shaders

The main practical challenge we had to overcome in this work is the fact the `Shader` dialect of SPIR-V disallows many essential language features that are needed to compile languages such as C++. We structure our discussion of the SPIR-V `Shader` dialect as a back and forth between problems and solutions, accompanied with motivating examples in either C or SPIR-V.

We omit a formal description for each transformation (i.e., in the form of rewrite rules), and instead show their effect applied to the motivating example using their human-readable syntax. For further details on our implementation, we refer the reader to Appendix A as well as its source code.

5.1. Logical Pointers

Addressing models in SPIR-V describe how pointers can be created and used. The `Physical32` and `Physical64` models specify that pointers are *physical*: they have an observable bit-pattern. This allows reinterpreting memory, creating pointer-based data structures, and doing arbitrary pointer arithmetic. Unfortunately, `Shader` programs are currently restricted to the `Logical` model, which disallows all of this.

`Logical` pointers work similarly to C++ references: they cannot be offset or reinterpreted and while it is possible to select a sub-object, it is not possible to go back "up" from a sub-object to the parent object, even if they share the same memory location. Additionally, it is not legal to load or store a `Logical` pointer in memory, unlike C++ references. Listing 1 shows examples of all 3 problem scenarios.

Table 1: *Different types of memory on the GPU. The keywords used here are the ones used in Vcc.*

keyword	visibility
`global`	all threads in a dispatch
`shared`	threads within a work group
`private`	only available to one thread

These rules severely restrict what can be done with pointers and we devised two complementary approaches to overcome this problem.

5.1.1. Pointer Normalization and Demotion

Our primary solution is to normalize pointers to eliminate non-logical uses as much as possible. We wrote folding rules in our optimizer that move unsafe pointer casts and offsets to the *end* of the pointer computation. Loads and stores can inspect their pointer operand for bitcasts: we can rewrite a load from a cast pointer into loading a *different type*, and then casting the result, and likewise for stores. We also attempt to rewrite unsafe pointer offsets as safe sub-component accesses, and eliminate all redundant casts.

Our optimizer continuously checks the usage of variable declarations, if they have no uses they are eliminated, but if they only have logical pointer uses, we *demote* them to logical pointers that we can safely leave alone.

5.1.2. Memory Emulation

We are left the complex cases that our optimizer could not simplify away. We deal with them by creating arrays of a base type (`u32` by default) and lowering all pointers and memory accesses for a given address space to indices and load/stores into said array. We recursively deconstruct complex types into their components, and bitcast the individual words as we store them in the array. The arrays are sized so that they can contain all the variables they replace, and do not incur a significant memory penalty.

This approach gives us correct results for arbitrary pointer usage but is quite heavy-handed. Doing so obscures the original semantics of the program, notably because they remove explicit local allocations, as a result might make further optimizations difficult for the graphics driver. Recent versions of Vulkan do support physical pointers into *global memory* by enabling the `PhysicalStorageBuffer` feature, so we only apply this solution to `Private` and `Shared` memory.

5.2. Lack of Generic Address Space

On a GPU, there are three major types of memory we are concerned with, summarized in Table 1. When declaring memory, we must specify what address space it belongs to. This is usually done by using an extra keyword as an *address space qualifier* in the declaration.

```
int fib(int n) {
    if (n <= 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

int pow(int b, int e, int a) = 0 {
    switch (e) {
        case 0:  return a;
        case 1:  return b * a;
        default: return pow(b, e - 1, a * b);
    }
}
```

Listing 2: *Recursive (top) and tail-recursive (bottom) formulations of the `fib` and `pow` functions, respectively.*

Additionally, most compute APIs offer the possibility to use so-called "generic" pointers. These pointers can point into any memory type, negating the need to use address space qualifiers and therefore bringing the programming model closer to conventional CPUs, which only feature one address space.

Vulkan does not offer generic pointers, but emulating them is rather straightforward: We use the upper bits of the address as a tag, and replace each generic access with a function that looks up the tag and performs the appropriate access.

This tagging is well-defined on host architectures such as x86_64 because only the lower bits can be used in practice. Vulkan currently offers no such guarantees. However, we have experimentally found this to work across all the hardware we tested.

5.3. Lack of Recursion and Function Pointers

The SPIR-V specification disallows recursion, and does not feature function pointers. This means the recursive formulation of iterative algorithms is disallowed, regardless of whether they are tail-recursive or not. Listing 2 shows two examples of textbook formulations for simple recursive functions. Neither formulation is legal due to recursion.

Due to how we use Clang as a front-end we are able to benefit from the LLVM optimizer, which can eliminate tail-recursion for us. Therefore in our implementation work we focused on handling fully generic function calls.

5.3.1. SPIR-V is Effectively Stackless

The lack of recursion and function pointers lead to an interesting observation: It is perfectly possible for the implementation of a SPIR-V compiler to have no runtime function call support at all. Inlining is guaranteed to terminate, although program size might grow out of control.

This means it is also possible to implement local variables as static variables instead of allocating them on the stack. Likely because of this there is no support for dynamic stack allocation in SPIR-V, which meant we had to implement our own stack to support recursion and function calls. We simply reserve a runtime-configurable amount of bytes in `Private` memory and maintain our own stack pointers.

We observed that this scheme tends to blow up register pressure, as some compilers attempt to fit the stack in registers. As the program stack can get quite large, this easily blows past the register space and we saw poor occupancy. To mitigate this issue we added support for allocating the stack in `Global` memory and dividing it between subgroups at runtime. This approach saw good results as discussed in Section 6.

5.3.2. Emulating Function Calls

Due to the lack of function pointers in standard SPIR-V (There is an Intel exclusive extension for them [Int], available only in their OpenCL implementation.), we had to devise a scheme to emulate them outright.

First, we run an analysis to determine which functions are "leaf" functions, that is to say, functions that do not do indirect or recursive calls, and only call other leaf functions, transitively. As seen in Section 5.3.1, such functions can be implemented just by inlining recursively and do not require indirect jumps required for true function calls. Leaf functions do not require any further effort.

To emulate function calls we first lower them to tail calls: Non-leaf functions are split in two at the call site, and we spill and reload their live variables before and after the call respectively. We push the address of the latter half of the caller to the stack and tail-call the callee. Once the callee finishes, it pops the return address (the second half of the caller) from the stack and tail-calls into it to resume execution in the second half of the caller.

Now, we just need to emulate tail-recursion, which we do by surrounding the entire function in a switch case, then a loop. In every iteration, the switch "dispatches" functions calls which return the identifier of the next one to run. Doing so avoids static recursion: if we instead used a switch at each indirect call-site there would be a recursive path in the function call graph, which SPIR-V shaders disallow. We successfully implemented this "software scheduler" in Vcc and evaluated the performance in Section 6.

5.3.3. Lowering Function Calls to Callable Shaders

A somewhat recent development in Vulkan is the appearance of a hardware-accelerated ray tracing pipeline. These pipelines come with an interesting addition: Callable Shaders. Callable Shaders can be called indirectly, and recursively, and appear to be just specialized function calls.

We implemented an alternative function lowering pass that targets callable shaders, transforms function parameters, and returns into ray payload accesses. We map each unique function type to a global ray payload variable. The entry point of the shader becomes a ray generation shader, whilst every other callee becomes a callable shader.

Like the previous approach, this scheme requires some runtime support to setup the Shader Binding Table (SBT) appropriately. There are two issues with this approach:

- Despite the fact that callable shaders have a stack of their own,

(a) CornellBox **(b)** Bedroom **(c)** Dining Room

Figure 3: *Three scenes rendered with our pathtracer with a maximum ray depth of 5.*

we still need to use our emulated stack. Local variables are still bound by logical pointers rules, and so we need two parallel stacks: the one we emulate ourselves and the opaque one used by Vulkan. This is an unwanted complication and should be resolved at the API level.

- Ray tracing pipelines do not have workgroups or shared memory, and their subgroup composition (which threads run side-by-side) can be dynamically changed by the implementation for efficiency reasons. While this might be desirable for typical ray tracing materials, it causes divergence-sensitive operations to behave differently from the usual mental model.

5.4. Structured Control Flow

Just like C and C++, shading languages make use of structured programming [Dij79]. Structured programming organizes control-flow in a function in terms of high-level statements such as conditionals, pattern-matching, loops, etc. instead of naked jumps (`goto` and labels). While structured programming is standard practice in high-level languages, SPIR-V makes the interesting choice of requiring it in the intermediate representation, unlike other graphics APIs such as Metal or DirectX.

In SPIR-V, structure is expressed by applying annotations on certain branches in order to denote what structured constructs basic blocks belong to. These structured constructs correspond almost exactly with the control-flow statements available in GLSL. Selection constructs and loop constructs, corresponding to **`if`** and **`switch`**, and **`for`** and **`while`** loops in GLSL, respectively.

We will refrain from precisely describing the structured rules in full, for more details we refer back to the SPIR-V specification [KOK18] and the work by Klimis, Clark, Baker, et al. [KCB*23] on formalizing its structured control flow rules.

The presence of structural information in SPIR-V is justified by its utility when defining the semantics of non-uniform SIMT intrinsics. The recently released "Maximal Reconvergence" extension[Theb] uses the structured control-flow information to precisely define the set of active threads that should be converged at any point in the program.

In Shady, we rely on the conventional approach of running a structurizer [EH94] on unstructured programs to turn them into structured code. Our compiler IR is actually able to represent structured control flow natively, and our optimizations are non-destructive. This limitation is instead due to our front-end: LLVM is an unstructured representation which lacks structured control-flow information.

6. Evaluation

6.1. Methodology

We measured execution times on the GPU using the most accurate methods available: For Vulkan, we make use of timestamp queries to write two time stamps in-between the compute pipeline dispatch. For CUDA, we use two CUDA events before and after the `cuLaunchKernel` call. We run each test program 5 times and report the median execution time.

6.2. Pathtracer

We implemented an interactive pathtracer supporting CPU, CUDA and Vulkan; sharing the same C++ rendering code for each supported device. The pathtracer utilizes its own BVH for scene traversal built in a pre-process step; and recursively computes the global illumination of a user given scene by estimating the incoming light using Monte Carlo integration on each bounce [PJH23]. We added next-event estimation (NEE) for area lights with multiple importance sampling (MIS) [Vea97], Hessian roulette and multiple material models (diffuse, conductor and dielectric). The conductor and dielectric material support roughness using the GGX microfacet model [WMLT07]. Support for area lights and environment light is also included.

In Figure 3 we showcase the three scenes used in evaluation. The modified CornellBox scene with rough dielectric and conductor boxes is using a single rectangular area light to illuminate the scene. The Bedroom scene is lit by a constant environment light and Dining Room is using three large area lights outside the camera frustum for illumination.

We showcase the performance results for all 3 scenes in Figures

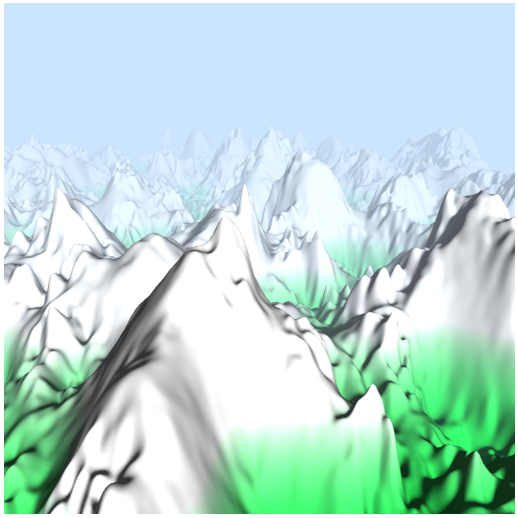

Figure 4: *This application uses Vcc shaders and achieves the same performance as with GLSL.*

Table 2: *Performance of our procedural terrain demo on various hardware configurations. Shown are median frametimes in milliseconds.*

Device	GLSL	Vcc
Radeon 7900 XTX	0.66	0.66
Geforce RTX 4070	13.40	12.66
Geforce RTX 3050Ti	3.94	4.00
i7-12700H (Xe 96EU)	13.67	14.20

5, 6 and 7. The Vcc numbers exist in three variants, referring to the normal compute shader mode, a special emulated scratch mode where we allocate the stack in a global memory buffer described in Section 5.3.1, and finally using callable shaders to implement function calls as described in Section 5.3.3 While CUDA tends to perform significantly better in the simple CORNELLBOX and relatively simpler DINING ROOM scene, the BEDROOM scene has them tied.

6.3. Realtime Procedural Geometry Demo

While we focused our efforts on the path tracer, we also did some evaluation work with the conventional graphical pipeline. We wrote a simple Vulkan application with GLSL shaders that renders procedurally generated terrain using Perlin noise, see Figure 4. We used a MIT-licensed shader from Inigo Quilez as the base [Qui]. We then ported the shaders to Vcc and measured the performance difference, presented in Table 2.

We found that the performance was virtually identical, which we attribute to the lack of complex features in those shaders. This does raise an interesting point: C++ shaders that do not rely on the features we covered in Section 5 simply compile down to something equivalent to their GLSL counterparts. Yet the benefits of code sharing and compile-time C++ features like templates remain.

6.4. Microbenchmarks

We wanted to have an idea what the overhead of the compilation techniques described in Section 5 was. To that end we wrote a small suite of microbenchmarks, designed to stress test particular components.

All presented benchmarks as shown in Table 3 are written using standard C++ with the language extensions described in Section 4.2. Our benchmarks use a common shared abstraction that handles device initialization and just-in-time compilation. This allows making the best use of Vulkan's many optional features and extensions.

6.4.1. Add Buffers

This benchmark does not stress-test any functionality in particular and instead serves as a baseline to validate that our setup is not adding any significant overhead between different APIs. The kernel adds two buffers together, one element per invocation. Performance is identical between Vcc and CUDA.

6.4.2. Unions

This benchmark showcases the ability to load and store different types to the same memory addresses. We define a type `S` that can carry either an `int` or a `float`, as well as a tag specifying how the data should be interpreted. We first copy an array of those types from global into private memory. We then mutate it and then hash the whole array using a hash function that works with bytes. This process is repeated 16 times.

The resulting performance numbers reveal that for NVIDIA cards, our approach of emulating untyped memory does not adversely affect performance compared to native CUDA. We tried this benchmark with two different word sizes for the emulated memory array: The `int8` path is significantly slower on the AMD card, however the NVIDIA hardware seems to not suffer much, if at all.

6.4.3. Binary Tree

This benchmark showcases the ability of our compiler to work with recursive data structures using generic pointers. On the host, we create a balanced binary search tree and fill it with a large number of elements. First, we upload the list of elements and then the tree to GPU memory. Afterwards, a kernel tasked with finding each element of the list in the tree is dispatched.

The binary tree is stored in global memory, but the pointer to it is *generic*. This means all of the accesses need to look at the tag in the upper bits of the address and perform a branch before the load can occur, as described in Section 5.2.

We observe similar performance to CUDA on Vulkan with the NVidia card. The 7900 XTX outperforms it significantly, possibly indicating additional driver optimisations.

6.4.4. Recursive Function Calls

The kernel computes the Nth number in the fibonacci sequence, for N = invocation ID modulo 16, because of the limited stack space available.

We observe that our implementation of function calls adds some

Table 3: *Median execution time of the microbenchmarks over 10 runs, in microseconds. The reference is obtained by running an equivalent CUDA program on the RTX 2080 Super. We used the radv driver for the AMD card.*

benchmark	Reference	2080S	7900 XTX
`add_buffers`	456	456	633
`unions (word=8)`	9147	8834	17143
`unions (word=32)`	9147	8548	4652
`binary_tree`	6009	7056	955
`fib`	311612	776337	328598
`fn_ptr`	644	14620	14225

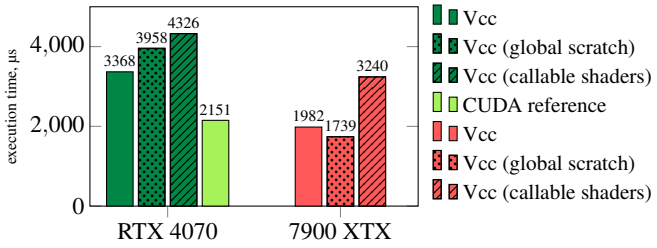

Figure 5: *Median execution time of our renderer for the* CORNELL-BOX *scene.*

significant overhead compared to CUDA. This is partly due to a lack of an appropriate mechanism to do indirect jumps: we had to pass all arguments and return values through the stack, every time.

6.4.5. Function Pointers

The function pointer benchmark selects one function out of four possible options and stores its address in a local variable, before calling it.

Like in the previous section, we note a considerable slowdown compared to the CUDA reference. It should be understood this is a worst-case scenario, synthetic test: each function body does essentially no work and the overhead of the function call is therefore maximized. It is also likely the native CUDA version benefits from some sort of specialization or interprocedural register allocation.

7. Conclusion

Our evaluation shows that our approach of compiling C/C++ is not far off and even occasionally competitive with CUDA, while

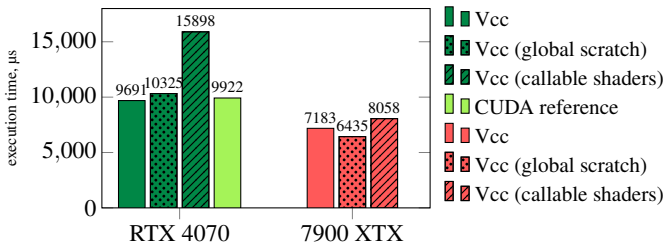

Figure 6: *Median execution time of our renderer for the* BEDROOM *scene.*

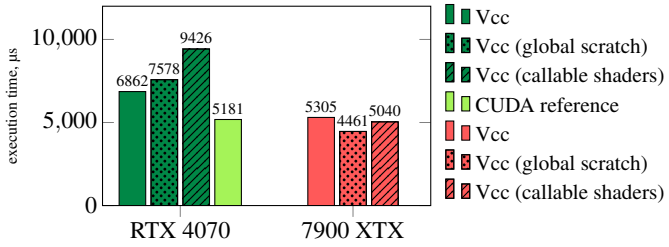

Figure 7: *Median execution time of our renderer for the* DINING ROOM *scene.*

retaining the portability of Vulkan. Performance does suffer with the use of certain features, with by far the worst results occurring when our emulated implementation of function calls is relied upon. Where this is not the case, we see that writing shaders in C++ can have no intrinsic cost at all (Section 6.3). The remaining issues we face are almost universally rooted in the technical limitations of our implementation.

7.1. Improvements to Vulkan/SPIR-V

We think of our strategies as a way to overcome the current limitations of SPIR-V shaders and therefore as a starting point for ecosystem improvements. In particular, allowing OpenCL features such as physical and generic pointers in shaders would negate the need for the techniques described in Section 5.1 and Section 5.2.

One of the most compelling improvements would be first-class support for function calls, to avoid the issues in Section 6.4.5. Since drivers have knowledge of the exact target architecture, their compilers can implement efficient calling conventions as well as inter procedural register allocation to reduce the number of spilled variables.

Our work could be practically useful, by offering proof of the viability of the concept we hope to motivate additional work. Additionally, because our current implementation does not require any more functionality than what exists today, it could accelerate adoption of native function calls by offering a viable software fallback.

7.2. Future Work

Our renderer did not implement support for Vulkan's built-in acceleration structures. In the future we plan to expand our renderer to include this support and explore the performance of our BVH in more detail.

Our work implements the basis of a C/C++ compiler for the GPU, but does not tackle the question of standard library support: functionality reliant on the host operating system such as `malloc` is unavailable. Implementing such functionality was deemed out of scope for this work.

Our compiler has the capability to emit compute programs but also shader stages for use in graphical pipelines. In this work, we discussed core limitations in shading languages from the perspective of general-purpose compute applications, but our techniques can be equally applied to graphical applications, however we did not evaluate it formally for such.

In Section 5.4 we kept our discussion of divergence-observing operations short, and our renderer currently does not use any subgroup intrinsics. This is partly due to how LLVM does not provide us with the structured control flow we would need to implement Maximal Reconvergence. Augmenting the C language family with such a model and implementing it in a GPU compiler should be a priority in future research.

Acknowledgments

This work is supported by the Saarland/Intel Joint Program on the Future of Graphics and Media, the Federal Ministry of Education and Research (BMBF) as part of the FAIRe project, and the Federal Ministry for Economic Affairs and Climate Action (BMWK) as part of the SolarEnvelopeCenter project.

We would like to thank our coworkers, students and friends who provided plentiful additional feedback before and after submitting the article. In particular, several Mesa contributors have been immensely helpful, both directly in a supportive manner, but also indirectly by making driver internals easily accessible for study.

References

[App06] APPEL, ANDREW W. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006. ISBN: 978-0-521-03311-4 10.

[App24] APPLE INC. *Metal Shading Language Specification*. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf. 2024 2.

[BBH*13] BRAUN, MATTHIAS, BUCHWALD, SEBASTIAN, HACK, SEBASTIAN, et al. "Simple and Efficient Construction of Static Single Assignment Form". *Compiler Construction: 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, 102–122 10.

[Dij79] DIJKSTRA, E. "Structured Programming". *Classics in Software Engineering*. USA: Yourdon Press, 1979, 41–48. ISBN: 0917072146 6.

[EH94] EROSA, ANA M. and HENDREN, LAURIE J. "Taming Control Flow: A Structured Approach to Eliminating Goto Statements". *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*. Ed. by BAL, HENRI E. IEEE Computer Society, 1994, 229–240. DOI: 10.1109/ICCL.1994.288377 6.

[Goo] GOOGLE. *google/clspv: Clspv is a compiler for OpenCL C targeting Vulkan compute shaders*. URL: https://github.com/google/clspv 3.

[HFF18] HE, YONG, FATAHALIAN, KAYVON, and FOLEY, THERESA. "Slang: Language Mechanisms for Extensible Real-Time Shading Systems". *ACM Transactions on Graphics (TOG)* 37.4 (2018), 141. DOI: 10.1145/3197517.3201380 3.

[Int] INTEL CORPORATION. *SPV_INTEL_function_pointers*. URL: https://github.com/intel/llvm/blob/sycl/sycl/doc/design/spirv-extensions/SPV_INTEL_function_pointers.asciidoc 5.

[Joh85] JOHNSSON, THOMAS. "Lambda Lifting: Treansforming Programs to Recursive Equations". *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. Ed. by JOUANNAUD, JEAN-PIERRE. Vol. 201. Lecture Notes in Computer Science. Springer, 1985, 190–203. DOI: 10.1007/3-540-15975-4_37 10.

[Kar] KAROL HERBST, MESA CONTRIBUTORS. *Rusticl - The Mesa 3D Graphics Library*. URL: https://docs.mesa3d.org/rusticl.html 3.

[KCB*23] KLIMIS, VASILEIOS, CLARK, JACK, BAKER, ALAN, et al. "Taking Back Control in an Intermediate Representation for GPU Computing". *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571253 6.

[Kes05] KESSENICH, JOHN. "Features of the OpenGL Shading Language". *White Paper* 3 (2005) 1, 2.

[Kir*07] KIRK, DAVID et al. "NVIDIA CUDA Software and GPU Parallel Computing Architecture". *ISMM*. Vol. 7. 2007, 103–104 2.

[KOK18] KESSENICH, JOHN, OURIEL, BOAZ, and KRISCH, RAUN. "SPIR-V Specification". *Khronos Group* 3 (2018), 17 2, 6.

[Lar19] LARSEN, STEFFEN HOLST. "Futhark Vulkan Backend". Master's thesis, University of Copenhagen, 01 2019, 2019 3.

[LBH*18] LEISSA, ROLAND, BOESCHE, KLAAS, HACK, SEBASTIAN, et al. "AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries". *Proceedings of the ACM on Programming Languages (PACMPL)* 2.OOPSLA (2018), 119:1–119:30. DOI: 10.1145/3276489 3.

[LLV] LLVM CONTRIBUTORS. *The LLVM Compiler Infrastructure*. URL: https://www.llvm.org 3.

[Mol11] MOLL, SIMON. "Decompilation of LLVM IR". *Bachelor's Thesis* (2011) 3.

[Mun09] MUNSHI, AAFTAB. "The OpenCL Specification". *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, 1–314 2.

[PJH23] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically based rendering: From theory to implementation*. MIT Press, 2023 6.

[PM03] PEEPER, CRAIG and MITCHELL, JASON L. "Introduction to the DirectX® 9 High Level Shading Language". *ShaderX2: Introduction and Tutorials with DirectX* 9 (2003) 1, 2.

[Qui] QUILEZ, INIGO. *Noise - simplex - 2D*. URL: https://www.shadertoy.com/view/Msf3WH 7.

[RL16] REYES, RUYMAN and LOMÜLLER, VICTOR. "SYCL: Single-source C++ accelerator programming". *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, 673–682 2.

[RLG*09] ROST, RANDI J, LICEA-KANE, BILL, GINSBURG, DAN, et al. *OpenGL Shading Language*. Pearson Education, 2009 2.

[Rus] RUST-GPU CONTRIBUTORS. *Rust-GPU*. URL: https://github.com/Rust-GPU/rust-gpu 3.

[Sam17] SAMPSON, ADRIAN. "Let's Fix OpenGL". *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by LERNER, BENJAMIN S., BODÍK, RASTISLAV, and KRISHNAMURTHI, SHRIRAM. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 14:1–14:12. DOI: 10.4230/LIPICS.SNAPL.2017.14 1, 2.

[TGF21] THOMAN, PETER, GOGL, DANIEL, and FAHRINGER, THOMAS. "Sylkan: Towards a Vulkan Compute Target Platform for SYCL". *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456683 3.

[Thea] THE KHRONOS GROUP. *KhronosGroup/SPIRV-LLVM-Translator/: LLVM/SPIR-V Bi-Directional Translator*. URL: https://github.com/KhronosGroup/SPIRV-LLVM-Translator 3.

[Theb] THE KHRONOS GROUP. *VK_SPV_shader_maximal_reconvergence*. URL: https://github.com/KhronosGroup/SPIRV-Registry/blob/main/extensions/KHR/SPV_KHR_maximal_reconvergence.asciidoc 6.

[Vea97] VEACH, ERIC. *Robust Monte Carlo methods for light transport simulation*. Stanford University PhD thesis, 1997 6.

[WMLT07] WALTER, BRUCE, MARSCHNER, STEPHEN R., LI, HONGSONG, and TORRANCE, KENNETH E. "Microfacet Models for Refraction through Rough Surfaces". *Proceedings of the Eurographics Symposium on Rendering Techniques, Grenoble, France, 2007*. Ed. by KAUTZ, JAN and PATTANAIK, SUMANTA N. Eurographics Association, 2007, 195–206. DOI: 10.2312/EGWR/EGSR07/195-206 6.

Appendix A: Implementation

In this section, we are going to share some more practical details on Vcc compiler, our compiler that transforms C/C++ code into SPIR-V shaders suitable for Vulkan. The implementation of the Vcc compiler as well as associated samples and benchmarks is available as open source software at `https://github.com/shady-gang`.

Features and Limitations of the Compiler

Vcc implements a significant subset of the C++ language, but leaves out certain aspects. Supported features include:

- Arbitrary control-flow inside functions, including `goto`.
- Arbitrary pointer usage for all memory types, including global, shared, and invocation-private. Pointers into I/O variables can be supported with extra copies. This also includes C++ features that are implemented with pointers in LLVM, such as references or bitfields. `memcpy` is supported.
- Arbitrary function pointers and calls within a shader module. Lambda functions work, but not `std::function`, due to STL issues. The function pointers are 64-bit and can be converted to integers, for example to implement sorting to combat divergence.
- Recursion, with user-defined stack size, as done in CUDA.

Vcc works for graphics and compute pipelines. We have also experimented in a limited fashion with ray tracing pipelines, as seen in Section 6.2.

Since we use an unmodified Clang front-end, the limitations are actually entirely dictated by the subset of LLVM IR we support. The following features were not implemented due to scope and time constraints:

- The C and C++ standard libraries. Some of their functionality would require some sort of remote procedure call into the host, which was not in scope for this work.
- C++ exceptions. They require implementing a complex stack unwinding mechanism, which would be an engineering-heavy task for a feature that many C++ programmers do not use.
- C++ virtual functions. Likewise doable, but sharing C++ objects from host and device would not result in portable vtables, which would make the feature a liability.

Finally, the following features are not planned as part of this work as they would require a different or at least complementary approach to tackle things beyond the scope of this work:

- Unified address space between host and device. Vulkan does not offer this feature and support would require software emulation.
- Single-source programming model. A tightly-integrated host language / runtime component would be required for launching GPU work from the host language.

Compiler Framework

Figure 8 shows the overall compilation flow of Vcc. We rely on Clang as a C/C++ front-end. Clang emits LLVM IR, which we convert to Shady—the IR and underlying compiler framework of Vcc.

Shady can represent GPU programs and supports the analysis and transformations our compilation pipeline requires. It can parse SPIR-V and LLVM IR, and can emit SPIR-V, or C-family languages such as CUDA, GLSL, or C11. Shady is not C or C++ specific and could easily support other languages.

Our representation is a hybrid between a Control-Flow Graph (CFG) and a structured representation. We support the typical **`jump`**, **`branch`**, and **`return`** terminators found in SPIR-V or LLVM and we also have a structured flavor using **`if`** and **`loop`** terminators that correspond to SPIR-V's structured control flow primitives.

Shader Compilation Pipeline

Most of the transformations are implemented as node rewriting functions, applied to the whole module at once. The broad pass order is as follows:

1. Continuation-Passing Style (CPS) transformation: direct-style calls are replaced with tail calls and return parameters are added [App06].
2. Closure conversion: basic blocks used as higher-order functions are lambda-lifted to top-level functions [Joh85].
3. Tail call lowering: Tail calls are eliminated and a top-level dispatch function is generated if required, as described in Section 5.3.2.
4. Generic pointer lowering: We generate the access functions described in Section 5.2.
5. Physical pointer lowering: We generate the global arrays and functions described in Section 5.1.
6. Restructuring: Non-structured control flow is eliminated from the IR: all control-flow is now done with **`if`** and **`loop`** as described in Section 5.4.

A cleanup phase runs after each pass, applying a simple SSA construction pass [BBH*13], inlining blocks used only once and demoting physical memory allocations to logical where possible. The clean process runs iteratively until a fixed point is reached.

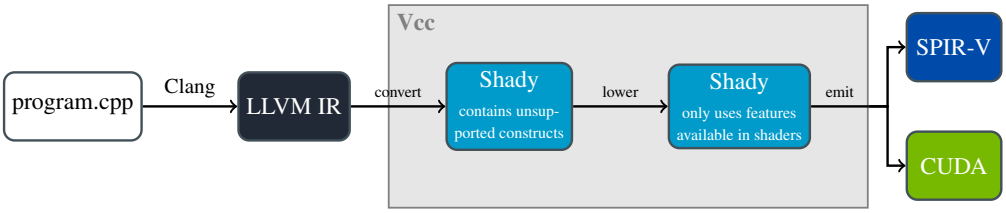

Figure 8: *Overall compilation pipeline: Vcc compiles shader programs written in C/C++ via Clang/LLVM to SPIR-V and CUDA. The Shady IR derived from the LLVM IR is only valid for SPIR-V after applying the lowering passes in Vcc.*