Vulkanised 2024

The 6th Vulkan Developer Conference
Sunnyvale, California | February 5-7, 2024

Vcc: Extending Standard C/C++ For Writing Vulkan Shaders

Hugo Devillers, Saarland University

Outline

- What is Vcc ?
- Why are we building this?
  - Commentary on shading languages
- How does this work?
  - Broad strokes of our approach
  - Deep dive: pointers & memory
- Practical details and conclusion.
  - Trying out Vcc yourself
  - Planned and ongoing future work

Vcc: The Vulkan Clang Compiler

- Vcc is a C and C++ compiler for Vulkan based on Clang
  - Actually, we only use the front-end. More on that later
- It aims to support these languages “fully”: no subsets!
  - Arbitrary control-flow: goto, function pointers, recursion (!)
  - Real pointer support: pointer arithmetic, casting, etc
  - Generic pointers, without an address space, like in OpenCL 2.1
- These features are traditionally absent from shaders
  - But compute APIs have them…
  - We think graphics APIs should too!

Caveats

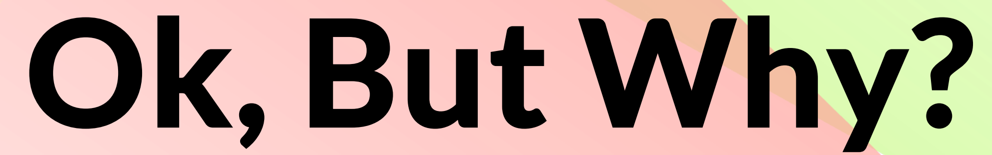
- This is a proof-of-concept:
  - No standard library: `-ffree-standing` mode only
  - No C++ exceptions, RTTI…
- This isn’t a SYCL/CUDA/HIP competitor
  - Just a shader compiler, not a single-source GPGPU platform
  - Not just compute: we also do graphical stages!
- We’re researchers, not full-time compiler vendors
  - Scope deliberately limited to make project viable
  - We’re interested in industry collaboration

Extending C++ for doing shaders

- We leverage Clang's existing extensions:
  - __ext_vector_type() for vector types
  - __attribute__((address_space(n))) for storage classes
- We use __attribute__((annotate(…))) to define:
  - Entry points and parameters (workgroup size …)
  - Pipeline layout information (set/binding, i/o locations)
  - Texture types and sampling operations
  - Planned: arbitrary SPIR-V intrinsics
- We #define all this in the <shady.h> header
  - It's a shading language defined as a library[1] !

hello_world.frag.cpp

```
#include <shady.h>
using namespace vcc;
location(0) output vec4 outColor;
extern “C” {
  fragment_shader void main() {
    outColor = vec4(1.0f, 1.0f, 0.0f, 1.0f);
  }
}
```


Ok, But Why?

State of Shading Languages

- GLSL and HLSL come from a very different era (20 years ago!)
  - Their initial design reflected compromises for then-current hardware
  - Hardware and graphics APIs are unrecognisably different now
- They significantly diverge from C and C++
  - Syntax-wise (writing ”hybrid” code with #defines is terrible)
  - Capability-wise (ban on recursion, no “real” pointers)
- Their evolution is slow and/or uncertain
  - Awkward additions (`buffer_reference` anyone ?)
  - Technical debt (HLSL is still stuck on Clang 3.x)
  - Lost of institutional knowledge when people retire !^{2}

Opinions

- Writing GPU code should not require specialized languages
  - Unecessary barriers to offloading work to the GPU in the first place
  - We can have graphics features on top of general-purpose languages
- Inexpressivity is not a shortcut to performance
  - Programmers who need something will use the next best thing
  - Hence, “straitjacket” languages may indirectly hurt performance
  - You can’t wish complexity away
- Instead, giving programmers expressive languages
  - Helps them express their domain knowledge
  - While keeping control over complexity

Escaping the Shader Islands

- The HPC world writes GPU code in C++
  - They get to share code meaningfully between host and device
  - Obviously performance matters just as much to those folk!
- Overall industry trending to do the same for graphics
  - HLSL 202x tries to bring more C++ features in HLSL
    - Slang, Circle occupy similar positions
  - Rust-GPU applies the same idea for Rust programs
- Maintenance: We could just be in mainline Clang
  - HLSL wants to get there as well, so this isn't a new idea

Why It's Not So Easy

- Vulkan uses a heavily restrictive dialect of SPIR-V
  - In line with the historical restrictions in GLSL
- "Normal" HLL compilers can't easily deal with:
  - The requirement of structured control flow
  - The lack of a stack and "real" function calls
  - The lack of physical pointers (we'll get to what that is)
- Chicken and egg problem:
  - High-level languages can't easily target Vulkan due to the restrictions
  - The Vulkan ecosystem needs motivating examples to drive change

Vcc: Under The Hood

Spoiler Alert: We made an IR.

Shady: A Futuristic IR For SPIR-V

- The key to this project is our “Shady” IR and compiler
  - It has everything we want: goto, pointers, function calls, a stack…
- Uses state-of-the art compilation techniques to compile it to a SPIR-V binary that Vulkan will be happy with
  - Similar to what e.g. clspv or rusticl have to do
  - With a focus on robustly dealing with worst-case scenarios
- Actually, we made Shady first …
  - Vcc is a (very fancy) tech demo for our research :-)
  - We encourage other languages to use it to target Vulkan more easily

The Big Picture

- Vcc calls into an unmodified version of clang
  - Gives it a bunch of flags: -O0, -free-standing, -isystem=...
  - Asks it to dump the generated LLVM IR, which shady can then parse
- No LLVM opts or transformations are involved
- We bring our own optimizer instead
  - Tries to remove anything "problematic"
- If anything unsupported remains, we emulate it
  - More details coming next
- Shady itself does not have any idea what C or C++ are
  - Although the IR obviously has semantics compatible with them

Case Study: Memory

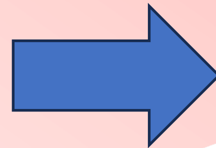
Dealing with Pointers

%0 = OpVariable %int

%1 = OpConstant %int 6

OpStore %0 %1

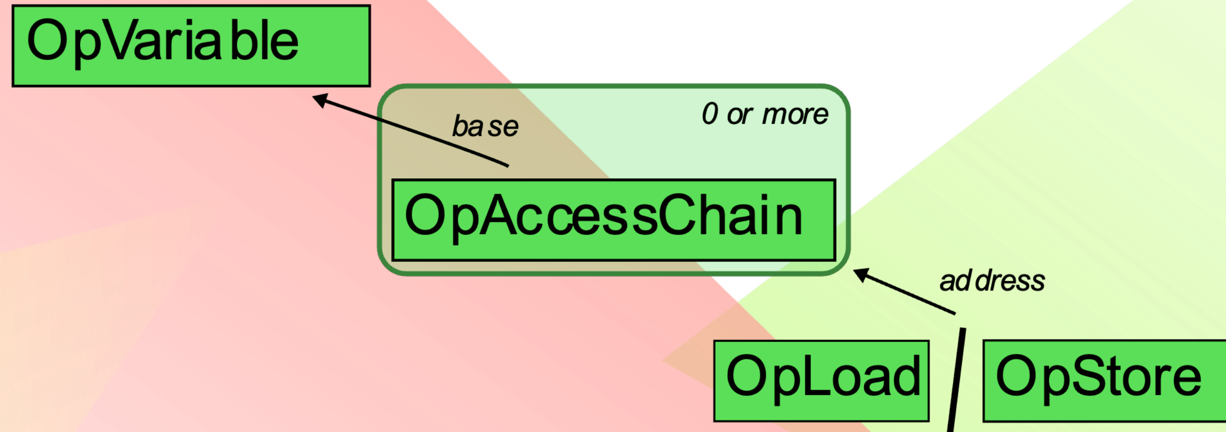
%2 = OpLoad %0



%1 = OpConstant %int 6

- Simplest solution: Optimise them away!
  - Shady implements SSA construction (mem2reg)
  - Gets rid of a lot of simple cases
- Doesn’t work for all cases

Vulkan SPIR-V Uses Logical Addressing



- OpVariable lets us allocate memory and gives us a pointer
- We can pick sub-elements using OpAccessChain
  - Equivalent to LLVM's GetElementPointer
- Pointer operands can always be "followed" back to an allocation

Logical Pointers Disallow …

OpLoad

address

OpLoad

- Loading or storing pointers themselves
  - Exceptions exist (variable pointers[3])
  - Complicated rules, bottom line unchanged

- Changing the pointee type
  - Must bitcast values instead

OpVariable f32

OpLoad

source

address

OpBitcast[*i32]

OpAddI

source

address

OpLoad

OpConvertUToPtr

- Casting to and from integers
  - Disallow untracable pointers to appear "out of thin air"

Legalizing Away Casts

- Some code can be transformed to fit logical addressing rules
  - Here a pointer cast is turned into a value cast

OpVariable f32

source

OpLoad

address

OpBitcast[*i32]

Legalizing Away Casts

- Some code can be transformed to fit logical addressing rules
  - Here a pointer cast is turned into a value cast

OpVariable f32

OpBitcast[i32]

address

OpLoad

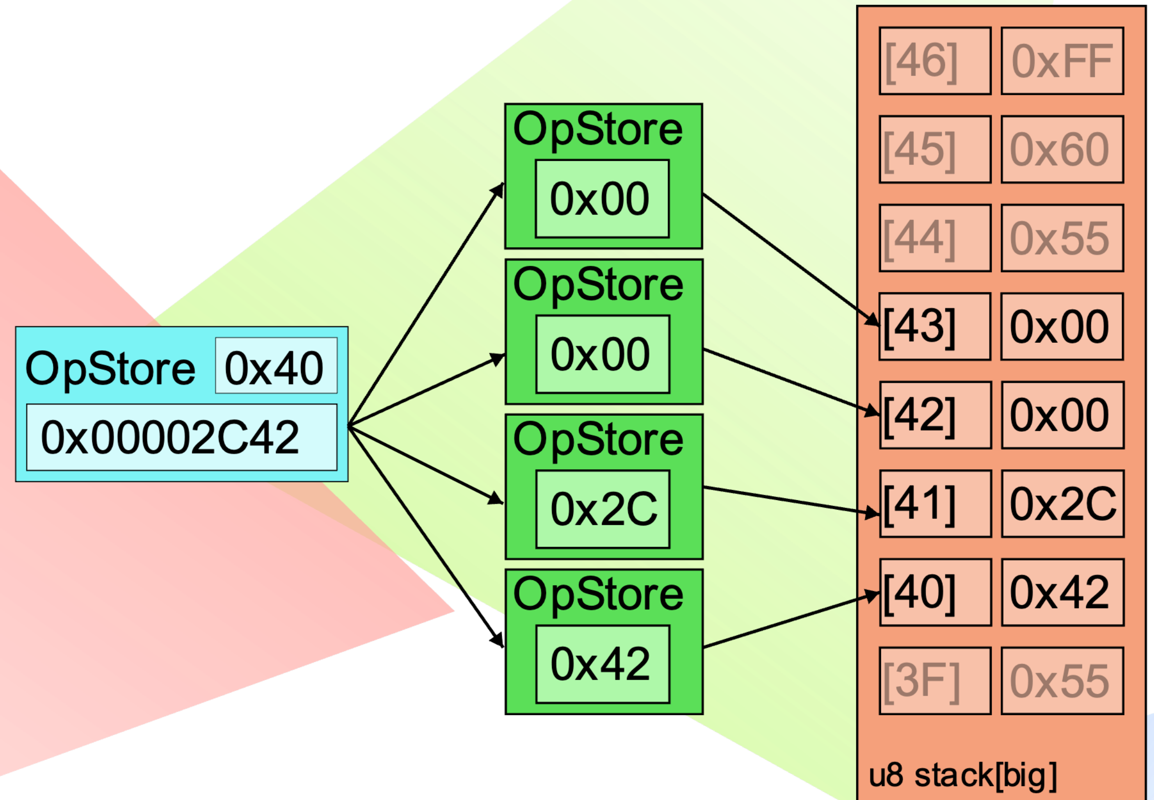
source

Consequences of Logical Addressing

- Pointers are "opaque", their representation is never observable
  - They might not be pointers at all in practice
  - Example: texture descriptors stored in hw registers
- Profoundly mismatched with C/C++ and LLVM pointers
  - High-level code translates to *a lot* of unsafe pointer arithmetic
- What we need here is Physical Pointers
  - Have a fixed-size representation and you can look at it
  - OpenCL has them (Physical[32/64] addressing mode)
  - Vulkan has PhysicalStorageBuffer, but for global memory only

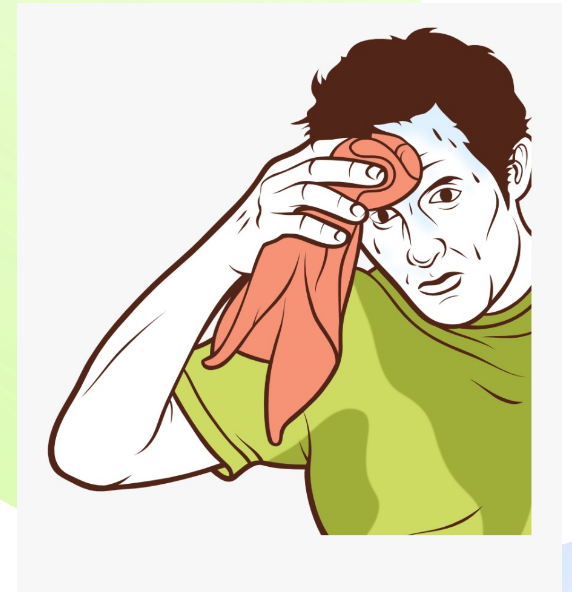
The Nuclear Solution

- We can *emulate memory* by storing everything into a big array of u8 words[4]
- Larger types are decomposed
- Reliable but very heavy-handed
  - We avoid this wherever possible
- Support for physical pointers directly would be a massive win
  - Most (all?) hw has this capability
  - Compute APIs all have it



Memory - Conclusion

- With global memory, things are easy:
  - `bufferDeviceAddress` gives us what we need
- We try to optimise aways allocations entirely
- We try to make memory accesses fit logical rules
- If all else fails, we fall back to the nuclear solution
  - We again plea for more physical pointers support !
- We employ a similarly complex set of heuristics to enable arbitrary control-flow
  - Bonus slides if time allows …



Layering Is Not Ideal, Actually

- As we’ve seen, Shady goes to great lengths to transparently implement “impossible” features given the constraints
- This is cool, but it’s far from ideal:
  - Unecessary complexity
    - Which shady handles for you, but it’s still paid for somewhere
  - Code size and general overhead
  - Semantic loss: the driver has fewer opportunities to optimise the result
- This is a great “interim” solution but can’t be relied upon forever
  - We hope to motivate the Vulkan ecosystem to push forward
  - In fact, we hope Shady is entirely redundant in a few years !

How’s Performance Now?

- Our emulation can be costly
  - No hard numbers yet, we need more applications
- Huge variability depending on the optimisations
  - We’re still working on our optimiser, there’s low-hanging fruit to pick
- Huge variability depending on the platform
  - ≠ shader compilers can make a massive difference on the same HW
  - GPUs with dynamic register allocation benefit dramatically from it
- Performance is comparable to GLSL/HLSL for programs that don’t rely on recursion or function pointers (ie: stack-less)

In Practice

Standing issues & Limitations

- Function calls currently rely on shared memory
  - Fixable by using subgroup communication directly intead
- Constants need to be provided in a buffer by the host
  - We can put constants in private memory, but it's very wasteful
- Some limitations can only be fixed at the ecosystem level
  - True shared libraries – need inter-module calls
  - Unified addressing – needs to be exposed by the driver
- We will be working within Khronos and with any interested vendors to ease the slow-paths
  - In particular when dealing with the stack and function calls

Future Work

- We'd like Shady to become redundant
- Here's how that might work:
  - Introduce extensions that let us use SPIR-V as our own binary format
  - Mainline LLVM can then target that, much like OpenCL SPIR-V
  - We can make use of the Vulkan layers system to have Shady provide assistance to drivers who can't support all the features natively
  - "When everyone is super, no one will be"
- There's probably room for a proper "C++ for Vulkan" dialect
  - Much like there is a "C++ for OpenCL" already.

Try Vcc Out Now !

- Project website: https://shady-gang.github.io/vcc
  - Windows binaries available
  - Documentation available there too
  - Compiler Explorer instance available
  - Samples (Vulkan-Tutorial, aobench, vcc-rt-demo)
- Written in pure C11, minimal dependencies:
  - Cmake, json-c, SPIRV and Vulkan headers
- Any version of LLVM from 14 onwards should work
  - A non-trivial amount of effort was spent on making that possible
- We welcome your feedback !

Questions

Footnotes & References

[1]: Our previous work has focused on domain-specific languages:

AnyDSL - A Partial Evaluation Framework for Programming High-Performance Libraries, https://anydsl.github.io/

[2]: I'd like to wish John Kessenich a long and pleasant retirement :)

[3]: Variable pointers are very narrow in the relaxations they offer. In the end we decided to ignore them since they would add complexity on our end with little to show for. We'd rather ask for a fully physical model.

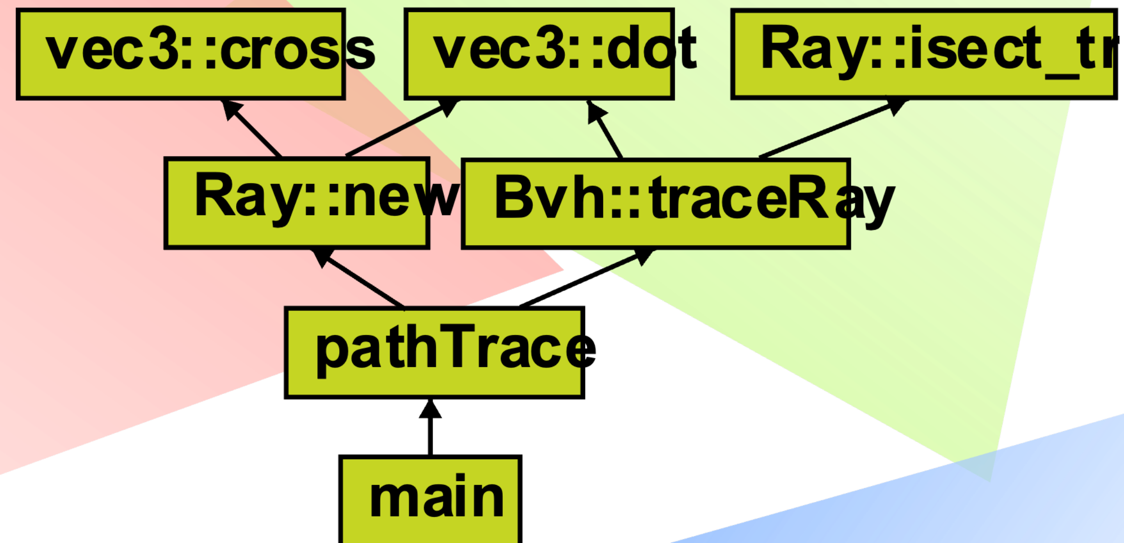
[4]: We can use other, larger base word types, but it makes accessing smaller data complicated and possibly unsafe (introduces concurrency hazards)

Bonus: Dealing with Function Calls

- We want arbitrary control-flow:
  - Arbitrary branching within a function
  - Function pointers and recursion
- Once more, legacy shading languages never supported this
  - SPIR-V can *express* them …
  - … But Vulkan disallows these things explicitly.
- Similar approach as with pointers:
  - Legalize what can be
  - Emulate the rest

Leaf Functions

- Leaf functions are functions that are only called directly, and don't call any function indirectly. They also can't be recursive.
- Leaf functions may (directly) call other leaf functions
- This is "easy mode"



Function Calls Example

```
%f = OpFunction %i32
%x = OpVariable %i32
OpStore %x 0
OpFunctionCall %g %x
%v = OpLoad %x
...
```

```
%g = OpFunction %void
%x = OpFunctionParameter %ptr_i32
OpStore %x 42
OpReturn
```

```
int f() {
    int x = 0;
    g(&x);
    …
}

void g(int* x) {
  x = 42;
}
```

Continuation-Passing Style

```
%f = OpFunction %i32
%x = OpVariable %i32
OpStore %x 0
OpStackPush %x
OpTailCall %g %f_ %x
```

```
%f_ = OpContinuation
%x = OpStackPop %ptr_i32
%v = OpLoad %x
...
```

- ”one-way” functions
- Split at call sites

```
%g = OpContinuation
%ret = OpReturnParameter ...
%x = OpFunctionParameter %ptr_i32
OpStore %x 42
OpTailcall %ret
```

- Locals pushed/popped in stack
- Return address parameter

Calls Recap

- We lower to continuation-passing style
  - Everything is a tail-call now
- We split functions in two when an indirect call is made
  - We spill all the live variables before the call and reload them after
  - We implement a stack in private memory to store spilled locals
- Issue 1: Vulkan doesn't have tail-calls
  - We emulate tail-calls with a global while loop.
- Issue 2: We spill and reload everything, all the time.
  - No control over register allocation/clobbering