Vulkanised 2024

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

Beyond SPIR-V: Single Source C++ and Shader Programming

(In Production!)

Mateusz Kielan, DevSH Graphics Programming
CEO & CTO
devsh@devsh.eu

What language is this?

```
uint16_t compileTimeCountLZero = mpl::countl_zero<uint16_t,6>::value;
uint16_t runTimeCountLZero = countl_zero(6);

constexpr bool A = is_integral<int>::value;

float32_t4 v; assert(alignof(v.x) == alignof(v));

#define IS_SAME(...) is_same<_VA_ARGS_>::value

const volatile float4 u;
static_assert(IS_SAME(float, remove_cvref<decltype(v.x)>::type));
static_assert(IS_SAME(remove_cv<decltype(v.x)>::type, add_lvalue_reference<float>::type));
static_assert(IS_SAME(remove_cv<decltype(v.x)>::type, add_lvalue_reference<remove_cvref<decltype(v.x)>::type>::type));

float x[4][4];
static_assert(IS_SAME(remove_extent<decltype(x)>::type, float[4]));
static_assert(IS_SAME(remove_all_extents<decltype(x)>::type, float));

static_assert(IS_SAME(make_signed<int64_t>::type, make_signed<uint64_t>::type));
static_assert(IS_SAME(make_unsigned<int64_t>::type, make_unsigned<uint64_t>::type));
#undef IS_SAME
```

What language is this?

```
int Q[3][4][5];
static_assert(3 == (extent<decltype(Q), 0>::value));
static_assert(4 == (extent<decltype(Q), 1>::value));
static_assert(5 == (extent<decltype(Q), 2>::value));
static_assert(0 == (extent<decltype(Q), 3>::value));

float32_t a = numeric_limits<float32_t>::min;
a = numeric_limits<float32_t>::max;
a = numeric_limits<float32_t>::lowest;
a = numeric_limits<float32_t>::epsilon;
a = numeric_limits<float32_t>::round_error;
a = numeric_limits<float32_t>::infinity;
a = numeric_limits<float32_t>::quiet_NaN;
a = numeric_limits<float32_t>::signaling_NaN;

struct S {
    int x;
    int a(int) { return 1; }
};
static_assert(has_method_a<S, int>::value==e_member_presence::is_present);
static_assert(!has_method_a<S, float, float>::value);
static_assert(!has_method_a<S>::value);
```

Both!

```
int Q[3][4][5];
static_assert(3 == (extent<decltype(Q), 0>::value));
static_assert(4 == (extent<decltype(Q), 1>::value));
static_assert(5 == (extent<decltype(Q), 2>::value));
static_assert(0 == (extent<decltype(Q), 3>::value));

float32_t a = numeric_limits<float32_t>::min;
a = numeric_limits<float32_t>::max;
a = numeric_limits<float32_t>::lowest;
a = numeric_limits<float32_t>::epsilon;
a = numeric_limits<float32_t>::round_error;
a = numeric_limits<float32_t>::infinity;
a = numeric_limits<float32_t>::quiet_NaN;
a = numeric_limits<float32_t>::signaling_NaN;

struct S {
    int x;
    int a(int) { return 1; }
};
static_assert(has_method_a<S, int>::value==e_member_presence::is_present);
static_assert(!has_method_a<S, float, float>::value);
static_assert(!has_method_a<S>::value);
```

Hold onto your seats!

What I'll cover in this talk:

- The state of HLSL for targeting SPIR-V, its compilers, and changes in DXC's functionality since our 2023 talk
  - what minimal changes DXC could make in the short term to best close gaps between C++ and HLSL
  - how to keep the door open for another C++-like language to SPIR-V compiler
- Creatomg a C++-compatibility layer to execute snippets of HLSL as host code, and have binary compatible structures
- Including SPIR-V Headers and Boost Libraries into HLSL - because shader compile times just aren't long enough!
- "Look what you made me do!" - Proofs of Concepts made with C++11 Templates + C++20 Preprocessor + Inline SPIR-V
  - Such as a completely impractical implementation of Function calls and Recursion
- Gaps in the SPIR-V and Vulkan specifications making the most interesting mega-shaders, UB or impossible
  - Discussion of different tiers of function calls I'd like to see implemented in SPIR-V
- Our compiler and STL, packaged on Godbolt for you to play with!

The TL;DR State of C++-like GPU Languages in 2024

DISCLAIMER: Talk contains a lot of personal opinions about the state of projects from my own perspective. I'm not a maintainer of, or associated with any of them, and definitely NOT making statements on their behalf.

The TL;DR State of C++-like GPU Languages 2024

DXC DXIL Out

DXC SPIR-V Out

Clang-HLSL

The TL;DR State of C++-like GPU Languages 2024

DXC DXIL Out

DXC SPIR-V Out

Clang-HLSL

The TL;DR State of C++-like GPU Languages 2024

DXC SPIR-V Out

DXC DXIL Out

Clang-HLSL

Standalone Shaders in SYCL without a runtime*

SYCL on Vulkan For Compute Workloads

GLSL

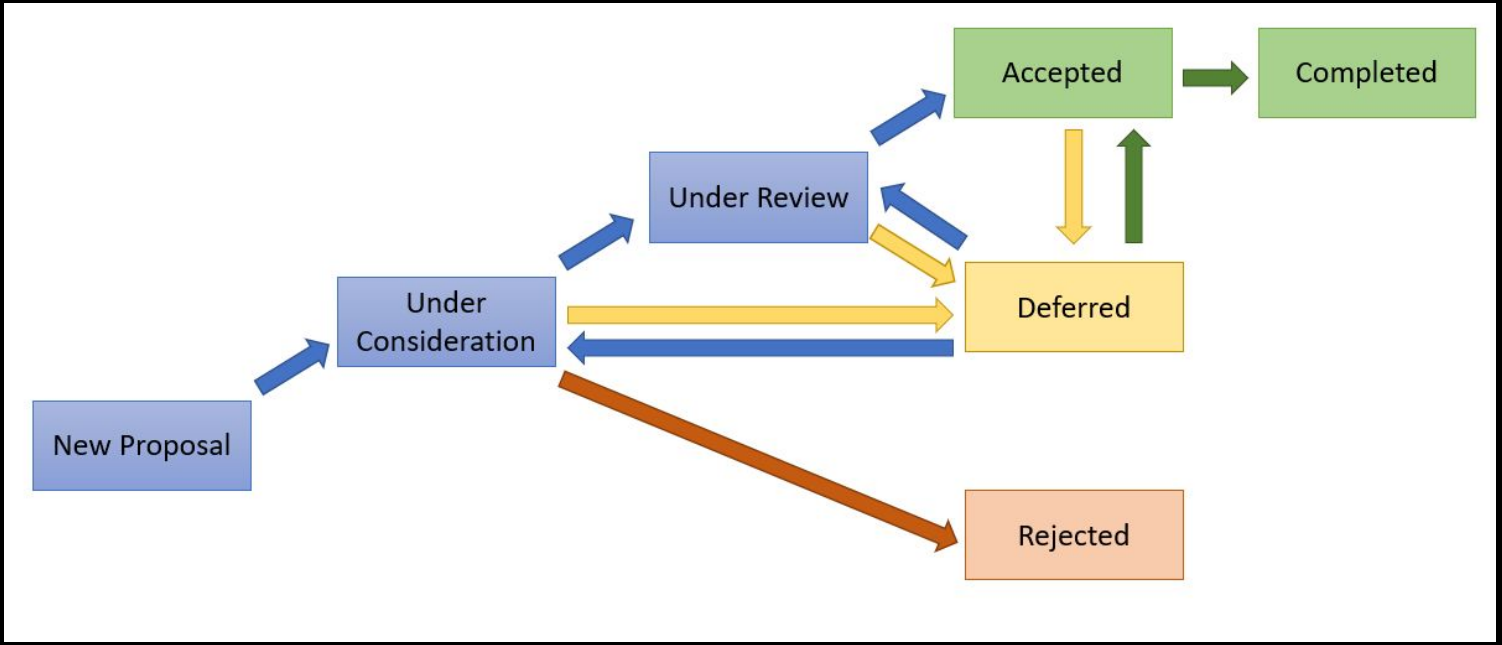
HLSL2021 with SPIR-V is actually production ready

The key is managing your expectations, take into account that:

1. DXC is a fork of Clang 3.7 which is more than 8 years old
    - Mainlining-HLSL-into-Clang effort on track, but everyone needs a maintained compiler in the meantime
2. the maintainers are very helpful and the community knows workarounds (*e.g. 64 bit image atomics*)
    - Microsoft does actually spend its own engineering resources on SPIR-V codegen
3. DXC's Inline SPIR-V feature fill gaps nicely, use it extensively
4. new **important** HLSL Language Proposals do actually get implemented in DXC, right now!
    - Examples: New Inline SPIR-V (0011) and Buffer Device Address (0010)
5. there is no alternative that's closer to C++ and equally tested across this much SPIR-V functionality
    - Can polyfill almost anything you don't like as long as Proposal 0011 gets implemented
6. everything else not included in (5) is held back by Proposal (0006)

What are HLSL Proposals? - Your chance to take part!

Per the 2022 announcement, the evolution of HLSL is open to the public: https://github.com/microsoft/hlsl-specs



Source: https://devblogs.microsoft.com/directx/opening-hlsl-planning/

Proposal 0006 - usual culprit why a feature can't be done

When DXC was created, a hacksaw was taken to Clang 3.7 and several things got removed, most importantly:

- references, pointers, both for variables and functions
- Abstract Syntax Tree serialization → no Precompiled Headers

so there's no notion of a reference or pointer in a particular address space, and a very weird calling convention.

Just because there's no `&` in the language, it doesn't mean the compiler isn't creating references under the hood:

```
template<class T> struct add_reference_helper<T> {
   static T member[1];
   using type = __decltype(member[0]); ← don't get your hopes up, it's not accounted for during codegen
};
```

You can actually see `&` in compilation errors of template functions for `out` and `inout` parameters.

HLSL has non-static member functions and operators, but we still can't stick a ` const` at the end of a declaration.

At the time of the HLSL2021 announcement, DXC supported more operators than it does currently, since then many operators got disabled → https://github.com/microsoft/DirectXShaderCompiler/pull/4135

Also the conversion operators like `operator bool()` or `operator T()` don't kick in, but unsure if it's the same cause...

Aren't `in`, `out` and `inout` references?

Question to the audience. What will be the value of `t`?

```
void Test(inout float a, inout float b) {
    b = 1.f;
    a = 2.f;
}

float t = 45.f;
Test(t, t);
```

HLSL DXIL: https://shader-playground.timjones.io/3027e53d28cbab2ff5e6a18e8d1d4753
GLSL SPIR-V: https://shader-playground.timjones.io/5de81522d7b5b90917090dbf60499e3b

Aren't `in`, `out` and `inout` references?

Question to the audience. What will be the value of `t`?

```
void Test(inout float a, inout float b) {
    b = 1.f;
    a = 2.f;
}

float t = 45.f;
Test(t, t);
```

It's UB!
(at least in GLSL)

HLSL DXIL: https://shader-playground.timjones.io/3027e53d28cbab2ff5e6a18e8d1d4753
GLSL SPIR-V: https://shader-playground.timjones.io/5de81522d7b5b90917090dbf60499e3b *

- Both use the value-return calling convention! https://github.com/llvm/llvm-project/pull/75397/files
- By accident, Glslang happens to produce the expected result here, until you swap the order of the assignments
  - GLSL specification is silent on the order in which parameters get copied out or in.
- DXC will make `t == 1` but only because it incorrectly represents `inout` as `& __restrict__`

Note that OpFunctionCall actually allows for the parameters to be variable pointers, this is how Glslang is able to get the modified `a` and `b` "out" of the function.* It's only GLSL and HLSL value-return causing the problem.

Wait, it gets worse!

You might be thinking “Wait? This compiles!?” - Yes, `in`, `inout`, `out` copy so they can perform implicit conversions!

```
int t = 14;
Test(t, t);
```

In the next example, there can be a 10x speed difference between the `buf` being a parameter and just a global.

```
struct Buf {
    float data[64];
};
void Swap(inout Buf buf, int i, int j) {
    float t = buf.data[j]; buf.data[j] = buf.data[i]; buf.data[i] = t;
}
```

If you stick to value-return, your performance is at the mercy of the SPIR-V Optimizer catching onto what you’re doing.

This is why we often use a templated stateful accessor pattern

```
template<Accessor>void Swap(inout Accessor buf, int i, int j) {
    float t = buf.get(j); buf.set(buf.get(i),j); buf.set(i,t);
}
```

Why do we need references?

Here you can see a crutch DXC has, an attribute you can put on a SPIR-V Intrinsic function parameter. I don't know if its still __restrict__ but either way, it only lets you pass a reference one "degree" away.

```
template<typename T>
T atomicAdd([[vk::ext_reference]] T ptr, uint32_t memoryScope, uint32_t memorySemantics, T value);
```

When we provide wrapped SPIR-V intrinsics with the intent of having GLSL compatibility, like this one:

```
template<typename T>
T atomicAdd(inout T ptr, T value)
{
	return nbl::hlsl::spirv::atomicAnd<T>(ptr,spv::ScopeDevice,spv::DecorationRelaxedPrecision,value);
}
```

the forwarding only works because DXC misrepresents an `inout` as `&restrict`

It will stop working when a PR that makes DXC implement value-return properly, gets merged.

With proposal 0011 it might be possible to "force reference-like-behavior" by constructing SPIR-V pointer types.

Proposal 0010 - Buffer Device Address

Current state of the `vk::RawBufferLoad<T>` and `vk::RawBufferStore<T>` API:

- Structs: Used to disallow, then silently produced bugs (needed to load/store members 1-by-1), now fails codegen
- Struct Member access performed by BufferLoad extracts from a temp load instead of building access chains
  - Assigning to `vk::RawBufferLoad<MyStruct>(addr).member` results in a compile error or a buggy
    - Not compatible with atomics due to not being a real l-value reference
  - Neither of the above relevant anymore, because `vk::RawBuffer` only works on POD types now
- Impossible to replicate the functionality of BDA with current Inline SPIR-V
  - Even if it could, you'd need Reflection on Structures to generate member accessors
    - My attempt, stunted by each new SPIR-V type needing an UUID: https://godbolt.org/z/EEoebnTPK

Proposal 0010, a great contribution by Greg Fisher of LunarG, fixes the above, and brings HLSL on par with GLSL
https://github.com/microsoft/hlsl-specs/blob/main/proposals/0010-vk-buffer-ref.md

Almost perfect in my opinion, we'll probably wrap the `vk::BufferPointer<S,A>` and change a few things, as it will:

- have the BufferPointer restrict-by-default, making for some very nasty footguns similar to those from last slide
- not provide ==, !=,bool() or arithmetic operators
- still no way to get the address back for a member variable: https://github.com/microsoft/hlsl-specs/issues/57
  - extremely tricky to do without Prop 0006

HLSL's Inline SPIR-V

To fully appreciate why Proposal 0011 rocks, you need to know how the current Inline SPIR-V sucks.

The most basic usage of Inline SPIR-V is to be able to generate the SPIR-V you want verbatim, e.g.

```
template<> [[vk::ext_instruction(spv::OpAtomicIAdd)]]
int32_t atomicAdd([[vk::ext_reference]] int32_t ptr, uint32_t memoryScope, uint32_t memorySemantics, int32_t value);
```

But you might also want to expose things other than functions:

- Extensions and Capabilities used `[[vk::ext_extension(_extension_name_)]]` `[[vk::ext_capability(_uint_code_)]]`
- Execution modes `vk::ext_execution_mode(_uint_code_, …);`
- New SPIR-V Types `[[vk::ext_type_def(UUID,_spv_type_enum)]] void createTypeUUID();` and `vk::ext_type<UUID>`
  - UUID is a constant literal, and you can't default UUID to the _spv_type_enum (OpType has params)
  - __COUNTER__ won't save you here, you need to "remember" the UUID you assigned
  - Can't template OpTypePointer declarations because the UUID needs to be assigned
  - Bonus: DXC has no clue what is the storage class is allowed on this type
    - Also no info on size and alignment, so the type can only be a Private or Function variable

The last two aren't attributes, they're functions which must be called from the entry point and reachable!

Proposal 0011 - Better Inline SPIR-V

The proposal: https://github.com/microsoft/hlsl-specs/blob/main/proposals/0011-inline-spirv.md

- Fixes pretty much everything on the previous slide
  - Much better syntax for declaring SPIR-V types, with:
    - Opaque Types `using my_opaque_t = vk::SpirvOpaqueType<uint32_t OpCode, ...>;`
    - Storable Types `using my_store_t = vk::SpirvType<uint32_t OpCode, uint32_t Size, uint32_t Alignment, ...>;`
    - Storage Space attribute for variable declarations `[[vk::ext_storage_class(uint32_t class)]]`
  - Execution modes are now attributes you can put on functions
- And adds extra features like support for declaring builtins, and stage inputs/outputs

Have you noticed that our listings use actual C++11 `spv` enums instead of hand-written literals?

Including `"unified1/spirv.hpp"` and `"unified1/GLSL.std.450.h"` (all 5000+ lines of it) in your HLSL, is great!

Just remember to surround the global operator overloads in the headers with `#ifndef __HLSL_VERSION` as DXC does not support global operator overloads yet.

We actually want to autogenerate complete HLSL2021 SPIR-V headers from JSON grammar ourselves, and will open a PR against SPIR-V Headers for everyone's benefit.

How to make your HLSL202x compile as Host C++

This is really powerful, as e.g. we can flip-flop between CPU and Compute clipping in n4ce without having separate codepaths!

- First Step, make a compatibility header! → In our case it is "nbl/builtin/hlsl/cpp_compat.hlsl"
- Use the Preprocessor! Pivot on __HLSL_VERSION, using anything other like !__cplusplus will lead to pain in the future
- The hard part is getting HLSL types and intrinsics in C++, and I've been looking for such a library for many years
  - Initially dismissed GLM due to no concern over bytewise compatibility, e.g. std140 and std430 layouts on the CPU

    But the `scalarLayout` Vulkan feature is a thing, so GLM is actually perfectly compatible now!
  - Force the same matrix memory layout, as 3x4 is the most used, in case of CPU SIMD we recommend row-major
  - The rest is just typedefs from GLSL to HLSL identifiers and "hiding" the GLM matrix type (and certain`operator*`)
- HLSL202x is actually sane and defines all `floatNxM`, etc. as aliases of `template matrix<T,N,M>`
  - Start using `floatB_t[N[xM]]`, `[u]intB_t[N[xM]]` where B=16|32|64 while you're at it ;)
- For `float16_t` grab yourself the Imath library from OpenEXR, it's actually so complete you can put it in a `glm::vec`
  - Obviously it's slower than `float32_t` on the CPU, so treat it more like a compression format.

How close can you get to Host code with SPIR-V ?

The year is 2024 and every Desktop GPU still getting driver updates from their IHV supports Vulkan 1.3 and SPIR-V 1.6

- Did my homework after Vulkanised 2023, made a Vulkan Nabla Core Profile by intersecting reports of all such GPUs except GCN and MoltenVK, and it turns out they all support:
  - 8, 16 and 64bit SSBO and UBO storage and integer math - but HLSL has no 8bit types
  - Buffer Device Address, so at least for Global Memory you get to have pointers
  - scalarLayout → DXC's `-fvk-use-scalar-layout` is a "all or nothing" flag, you can't decorate individual resources
- DXC always strips caps as SPIR-V legalization involves running SPIR-V Opt with at least the Capability Stripping pass
  - so your shader won't require Float64 just because you declared a `float64_t...` that's not statically used
  - Warning! SPIR-V Opt only strips unused capabilities on an explicit allow list!

So only `float64_t` is something that needs a capability to use, and `int8_t` needs Proposal 0011

It's a pain that Proposal 0006 blocks certain operator methods, because we could make the following transparent* types:

- `int8_t` in user-space indistinguishable from a native type
- `float64_t` in software on top of `uin64_t`

So what does this actually look like?

It's somewhat unfortunate that we still need to use the preprocessor, but this is how we do it:

```
#ifndef __HLSL_VERSION
#include <type_traits>
#include <bit>

#define NBL_CONSTEXPR constexpr
#define NBL_CONSTEXPR_STATIC constexpr static
#define NBL_CONSTEXPR_STATIC_INLINE constexpr static inline
#define NBL_CONST_MEMBER_FUNC const

#define NBL_REF_ARG(...) typename \
nbl::hlsl::add_reference<__VA_ARGS__>::type
#define NBL_CONST_REF_ARG(...) typename \
nbl::hlsl::add_reference<std::add_const_t<__VA_ARGS__>>::type

#else

#define NBL_CONSTEXPR const static
#define NBL_CONSTEXPR_STATIC_INLINE const static
#define NBL_CONST_MEMBER_FUNC
// As long as DXC does not "fix" the value-return calling convention
#define NBL_REF_ARG(...) inout __VA_ARGS__
#define NBL_CONST_REF_ARG(...) const in __VA_ARGS__

#endif
```

```
#if (__cplusplus >= 202002L && __cpp_concepts)

#define NBL_CONCEPT_TYPE_PARAMS(...) template <__VA_ARGS__>
#define NBL_CONCEPT_SIGNATURE(NAME, ...) \
        concept NAME = requires(__VA_ARGS__)
#define NBL_CONCEPT_BODY(...) { __VA_ARGS__ };
#define NBL_CONCEPT_ASSIGN(NAME, ...) \
        concept NAME = __VA_ARGS__;
#define NBL_REQUIRES(...) requires __VA_ARGS__

#include <concepts>


#else


// No C++20 support. Do nothing.
#define NBL_CONCEPT_TYPE_PARAMS(...)
#define NBL_CONCEPT_SIGNATURE(NAME, ...)
#define NBL_CONCEPT_BODY(...)
#define NBL_REQUIRES(...)

#endif
```

Introducing the Nabla HLSL Template Library

We've made quite some progress since 2023, we now provide a set of utilities as a Header Only HLSL Library, here's some:

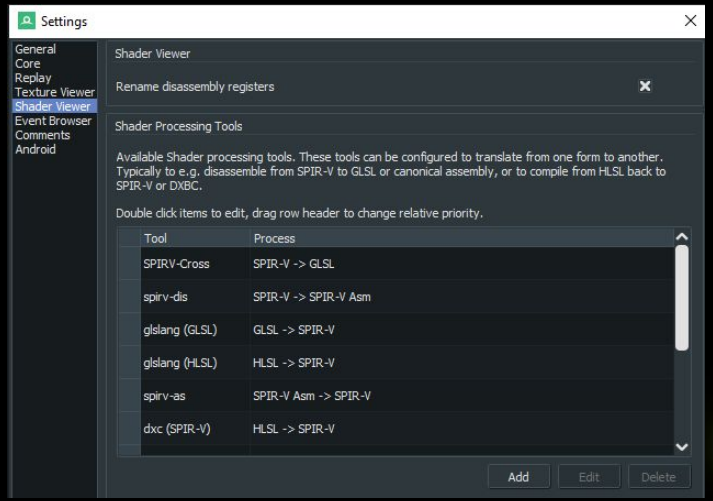
- Boost Preprocessor Library as an include and available in your HLSL or C++
  - Could include Boost MPL if HLSL had constexpr and refs, had to make our own impl. based on `integral_constant`
- SPIR-V Headers as an include (*can use all the enums in Constant Evaluated Contexts!*) and HLSL Inline SPIR-V Headers:
  - these have functions that propagate extensions, capabilities and execution modes; after proposal 0011, types too!
- Work-in-Progress: GLSL Compatibility Headers, providing `gl_` variables and GLSL built in functions
  - Very useful for things HLSL doesn't have, like 64bit image atomics, or fp64 `sqrt` and `length`
- The Cherry on Top; the reimplementation of relevant parts of the C++ STL in HLSL, still usable from C++!
  - Full and Complete <type_traits> → *that was the code from the first slide!*
  - <limits> adjusted for the fact that HLSL offers no constexpr-functions, `static const` inline members instead
  - <algorithm> heavily adapted to HLSL's lack of references → lack of iterators which can be assigned to
  - Work-in-Progress: <functional>, <tgmath> and <complex> in the order from most to least complete

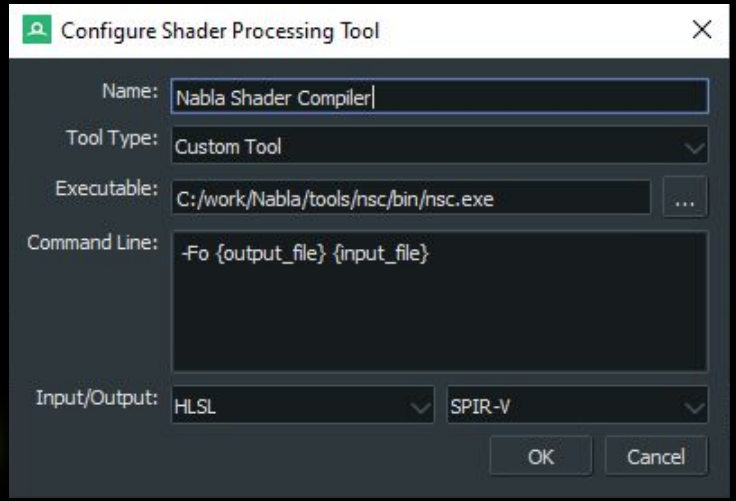
Introducing the Nabla Shader Compiler!

We made a utility on top of DXC targeting only SPIR-V 1.6 with some of the ubiquitous extensions. Created it cause we:

- used Godbolt extensively for Rapid Prototyping, grew tired of manually pasting all our Built-in Headers
- do all Preprocessing by ourselves, so macros, etc.get lost in debug info/source
  - Impossible to properly recompile with just stock DXC
  - Might need to fork from DXC over the reference/inout issue, need a HLSL->SPIR-V tool for Renderdoc
- sometimes just want to use an extra built in header while editing, that was no used by the original shader

Has a #pragma wave dxc_compile_flags() which helps us recompile the source with the same flags as the original.





Live Demo! (TODO)

Try it at https://godbolt.nsc.devsh.eu and compare with regular DXC (trunk) godbolt!

> DEMO_LONG_LINK → http://tinyurl.com/mrxnjnwj

No short links yet, working on it.

C++11 Templates + C++20 Preprocessor = Infinite Power!

You can actually preprocess your shaders independently of your shader compiler, be it DXC, Clang or glslang.

We can tell you "just use Boost.Wave" as it's probably the only conformant standalone library, we tried 2 others so far:

- Shaderc/glslang have no Variadic Macros and some bugs
- DXC has no way to hook up a virtual filesystem to provide `#include`s or support for __VA_OPT__
- Header-only, "light weight" libraries have trouble with basic Preprocessor logic

Why am I talking about a Preprocessor when I seem to be hailing templates as the replacement? Look at this:

C++11 Templates + C++20 Preprocessor = Infinite Power!

You can actually preprocess your shaders independently of your shader compiler, be it DXC, Clang or glslang.

We can tell you "just use Boost.Wave" as it's probably the only conformant standalone library, we tried 2 others so far:

- Shaderc/glslang have no Variadic Macros and some bugs
- DXC has no way to hook up a virtual filesystem to provide `#include`s or support for __VA_OPT__
- Header-only, "light weight" libraries have trouble with basic Preprocessor logic

Why am I talking about a Preprocessor when I seem to be hailing templates as the replacement? Look at this:

```
#define GENERATE_METHOD_TESTER(x) \
namespace nbl { \
namespace hlsl { \
namespace impl { GENERATE_STATIC_METHOD_TESTER(x, 4) } \
template<typename T BOOST_PP_REPEAT(4, NBL_TYPE_DECLARE_DEFAULT, 4), class=void> \
struct has_method_##x : false_type {}; \
BOOST_PP_REPEAT(4, GENERATE_METHOD_TESTER_SPEC, x) \
}}
```

Since C++ Reflection and Commercial Nuclear Fusion are eternally "just around the corner", the Boost Preprocessor Macros are the best way to auto-generate structures with reflection easily, for this __VA_OPT__ is particularly useful.

Runtime Specialization without Macros

Due to having a Virtual Filesystem, we require that any Runtime Compilation be a “Unity Build”.
This means that any #include must already be found and resolved, which interacts with the preprocessing logic.

As we used the preprocessor to specialize for the Runtime GPU Features and Limits, this introduced a conflict.
Our “solution” was to just ignore all preprocessing logic* and let includes recursively include themselves up to K times.

Obviously this doesn’t scale well, SPIR-V Headers are ~5k LoC, disabling the guards inflated simple shaders to 250k
Then the thing that broke this approach was Boost Preprocessor Library which relies on complex recursive includes.

We know that the next compiler we’ll target will be based on a recent version of Clang, so should expect:

- regular Pre-Compiled Headers, or
- C++20 Modules

all of which work on a serialized Abstract Syntax Tree (AST) that is produced after parsing the Post Processed C++.

TL;DR We can now no longer use macros and preprocessor defines to adapt our code to GPU Capabilities.

So how do we specialize now?

Device Capability Traits!

Remember the `GENERATE_METHOD_TESTER(x)` from a previous slide? We also have `GENERATE_MEMBER_TESTER(x)` too!

```
#define GENERATE_GET_OR_DEFAULT(field, ty, default) \
template<typename S, bool = has_member_##field<S>::value> struct get_or_default_##field : integral_constant<ty,S::field> {}; \
template<typename S> struct get_or_default_##field<S,false> : integral_constant<ty,default> {};
```

Here is a “trait” class that forwards static const members of the type you pass in, if its present, else defaults them

```
namespace impl
{
	GENERATE_GET_OR_DEFAULT(shaderFloat64, bool, false);
	GENERATE_GET_OR_DEFAULT(subgroupArithmetic, bool, false);
}

template<typename device_capabilities> struct device_capabilities_traits {
	NBL_CONSTEXPR_STATIC_INLINE bool shaderFloat64 = impl::get_or_default_shaderFloat64<device_capabilities>::value;
	NBL_CONSTEXPR_STATIC_INLINE bool subgroupArithmetic = impl::get_or_default_subgroupArithmetic<device_capabilities>::value;
};
```

You get to specialize without introducing any global state, and don’t interfere with AST serialization when it comes

```
template<class Binop, class device_capabilities=void>
struct inclusive_scan : impl::inclusive_scan<Binop,device_capabilities_traits<device_capabilities>::subgroupArithmetic> {};
```

Recursion with Macros and Templates

So apparently, you can't have recursion in HLSL,

Recursion with Macros and Templates

So apparently, you can't have recursion in HLSL, seems not → https://godbolt.org/z/r8dd9PjvE

```
#define SHADER_CRASHING_ASSERT(...) /* impl same as Sawicki*/

template<uint32_t MaxDepth = DefaultMaxRecursionDepth>
int fib(int n) {
	if (n < 3)
		return n;
	return fib<MaxDepth - 1>(n - 1) + fib<MaxDepth - 1>(n - 2);
}

template<> int fib<0>(int n) {
	SHADER_CRASHING_ASSERT(false); return 0xdeadbeef;
}
```

Please don't do this at home, the divergence and codegen are awful. I only thought about if I could, not if I should. This:

- only compiles with `-O0`, anything above and including `-O1` will murder SPIR-V Opt with exponential inlining workload
- generates 789 SPIR-V Opcodes with recursion depth 30, lower depth and `-O3` can generate as much as 1 MB of SPIRV

So let's talk about what I will do to work around the lack of function pointers / callables in non-RT stages, see this pseudocode:

```
#define REGISTER_FUNCTION(ID,RETVAL,.../*SIGNATURE*/) // horrible template specializations masquerading for reflection

function_ptr<RETVAL,Args...> p = CREATE_FUNCTION_PTR(ID);
p(args...);
```

The only I am missing is automated generation of the `function_ptr::operator()(Args...)` definition and the switch inside it.

How about we just have function pointers in SPIR-V?

It's not like GPUs can't do Function Calls or at least Subgroup Uniform Jumps to Arbitrary addresses:

- If the GPU implements Raytracing Pipeline and Callables, it must be able to call into functions from a table
  - Granted, it's a semi-manually managed stack (callable payload) but at least it's something
- Nvidia GPUs can do even more complex calls with regular stacks for over a decade in CUDA
- AMD could do it since GCN days, as demonstrated by Tomasz Stachowiak with glGetProgramBinary

You can of course split functions into call-with-continuation, and emulate function pointers via a God-dispatcher-loop containing a switch or hard-coded if-else binary tree search. But do we really want people to have to do that?

I've made a suggestion to Vulkan's SPIR-V: https://github.com/KhronosGroup/Vulkan-Docs/issues/2232

This could be approached incrementally, probably don't need full stackful function calls right away, they could:

- require function pointers to be subgroup uniform when calling
- introduce a Device Limit which tells you the maximum stack size (0 for no recursion allowed - only tail calls)
- disallow function pointer arithmetic
- disallow function pointer type reinterpret casting (like strict C or WASM)
- have a fallback layer that does exactly that looping God Function on the compiler level

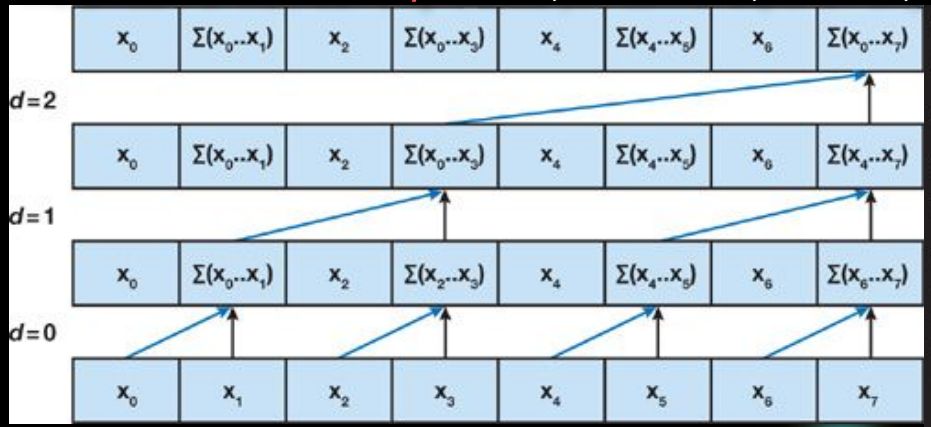
Next speaker's thoughts on the subject →

Inter-Workgroup sync - dangerously close to UB!

Suppose you have an algorithm where you want one workgroup to “consume” the outputs of another, for example my favourite party-trick, a reduction or the AMD Single Pass Downsampler.

The line between UB and not UB is very fine, as it stands the algorithm MUST be lockless.

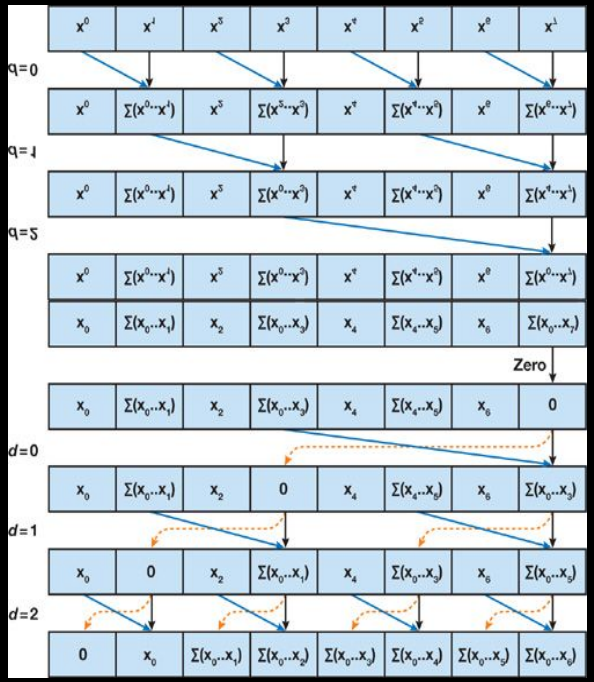
For example if you have N workgroups producing outputs for a dependant workload, this can be done by allowing any of the INITIAL producers of the prerequisites to become a consumer if all outputs are ready. *
The amount of work can only reduce, you cannot expand / amplify. Why is that?



https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

```
GenerateMIP1();
GenerateMIP2();
GenerateMIP3();
GenerateMIP4();
GenerateMIP5();
GenerateMIP6();
IncreaseAtomicCounter();
If (atomicCounter == numberOfThreadGroups)
{
    // Repeat above with offset 0
}
```

Inter-Workgroup waits - a trip to UB-land



https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

In SPIR-V there's no way to yield-wait on a condition or address, and spinlocks that require progress from another actor are UB!

They might work in practice*, but there's no contract which guarantees forward progress. E.g. WDDM has 3 different levels of preemption, the finest** can preempt mid-dispatch.

Imagine a "Deadlock Empire" style adversarial scheduler which keeps on picking the spinning subgroup or workgroup to execute instead of the one that supposed to make progress.

Consoles have forward progress guarantees for dispatches which just about saturate the GPU. I've opened a Vulkan issue about this and showcased the need with our Single Dispatch Scan. UE5 apparently also tried to spinlocks on PC, could be fixed in many ways:

- A new SPIR-V OpYield or OpWaitOnAddress
- Specifying how workgroups are scheduled

What I'd want DXC to do in the very near short-term

- Constexpr Function References `constexpr Return_t(&)(Args…)` to allow functions as template parameters
  - As long as you don't `export`, the function address is a compile time constant
  - No actual runtime logic or pointers → requires no new functionality in SPIR-V or DXIL
  - Would allow us to test for member function presence, exactly
    - Our test only checks that a member with identifier F can be called with arguments of type Args...
      - Remember `inout` allowing for implicit conversions?
    - It's not just about the ergonomics of not having to use a Struct type with operator() for a template
- Variadic Templates for tuples, and easy function parameter forwarding, required to finalize <type_traits>
- `constexpr` and `constexpr` functions, this is why our numeric_limits diverge from C++'s STL
- `alingof` when `-fvk-use-scalar-layout` is used, we currently have to refer to ancient C++03 style reflection tricks

```
#define NBL_REGISTER_OBJ_TYPE(T, A) namespace nbl { namespace hlsl { \
    namespace impl { template<> struct typeid_t<T> : integral_constant<uint32_t,__COUNTER__> {}; } \
    template<> struct alignment_of<T> : integral_constant<uint32_t,A> {}; \
    template<> struct alignment_of<const T> : integral_constant<uint32_t,A> {}; \
    template<> struct alignment_of<typename impl::add_lvalue_reference<T>::type> : integral_constant<uint32_t,A> {}; \
    template<> struct alignment_of<typename impl::add_lvalue_reference<const T>::type> : integral_constant<uint32_t,A> {}; \
}}
```

Questions ?

Additional Lovecraftian Horror - Pointer and Reference meta-wrapper-types:

- Indexing arrays of ByteAdressBuffers with MSB of 64bit pseudo-address → https://godbolt.org/z/KesGq3z44