Shading Languages Symposium 2026

Instrumenting SPIR-V for GPU-AV Validation

Spencer Fricke, LunarG

Who is Spencer

- Technical Lead of Validation Layers
  - Wrote most of the runtime SPIR-V Validation
- Have added a lot code to spirv-val (SPIRV-Tools)
- Wrote/Maintain SPIR-V Visualizer
- Wrote/Maintain SPIR-V Guide
- Help (sorry if slow) maintain SPIRV-Reflect

What you are about to witness

- How we designed our SPIR-V framework
- The evolution of the tool (from a SPIR-V perspective)
- Compromises we had to make
- Lessons learned
- Challenges that still plague us

Purpose of this talk

Quick background what GPU-AV is/does

What are the Vulkan Validation Layers

- "VVL" for short
- Tool to debug your Vulkan application
- The "layer" sits between the application and driver

What is GPU-AV

- GPU Assisted Validation
- Optional setting inside the Validation Layers
- Will try and validate things on the GPU at execution time

What is GPU-AV

- GPU Assisted Validation
- Optional setting inside the Validation Layers
- Will try and validate things on the GPU at execution time

https://www.youtube.com/watch?v=UeWXrOi7eBY

LUNARG

Overview of how GPU-AV works

1. Detect the SPIR-V being passed to the driver

Overview of how GPU-AV works

1. Detect the SPIR-V being passed to the driver
2. Instrument it with our own SPIR-V
    a. Will do some logic/validation check
    b. Will read/write to an "internal VkBuffer" we allocated

Overview of how GPU-AV works

1. Detect the SPIR-V being passed to the driver
2. Instrument it with our own SPIR-V
    a. Will do some logic/validation check
    b. Will read/write to an "internal VkBuffer" we allocated
3. Override the API calls to add our "internal VkBuffer"
    a. Done at a per-VkCommandBuffer granularity

Overview of how GPU-AV works

1. Detect the SPIR-V being passed to the driver
2. Instrument it with our own SPIR-V
    a. Will do some logic/validation check
    b. Will read/write to an "internal VkBuffer" we allocated
3. Override the API calls to add our "internal VkBuffer"
    a. Done at a per-VkCommandBuffer granularity
4. After vkQueueSubmit is resolved check if errors occurred

Goals of GPU-AV

- Detect things the spec says are invalid
- Be fast
  - VERY easy to turn 30 FPS to 30 seconds per frame!
- Provide a good error message
- Be simple to maintain the SPIR-V code
  - We don't want to maintain a compiler

With that, let's dive in!

LUNARG

Warning: Requires basic knowledge of SPIR-V

Original implementation used spirv-opt

- Updating a dependency was a painful extra step
- Hard to pass extra information
  - ex VkPipelineLayout
- Decided only way forward was do the passes inside VVL

Time to create our own SPIR-V framework

- Had existing code for normal (CPU based) SPIR-V validation
- The code was read-only

Time to create our own SPIR-V framework

- Had existing code for normal (CPU based) SPIR-V validation
- The code was read-only
- To instrument SPIR-V, we need to read-then-write
  - Much different challenge

Time to create our own SPIR-V framework

- Had existing code for normal (CPU based) SPIR-V validation
- The code was read-only
- To instrument SPIR-V, we need to read-then-write
  - Much different challenge
- Decided to have separate code
  - Still believe this was the correct decision

The following is how we designed our SPIR-V framework

Instruction class

- Designed to be a light wrapper around a SPIR-V instruction
  - Only stores the minimal data needed
- Generate finding the Result ID and Result Type ID

Debugging is no fun when data is not saved

```
uint32_t Length() const { return words_[0] >> 16; }
uint32_t Opcode() const { return words_[0] & 0x0ffffu; }
```

Debugging is no fun when data is not saved

```
uint32_t Length() const { return words_[0] >> 16; }
uint32_t Opcode() const { return words_[0] & 0x0ffffu; }
```

```
#ifndef NDEBUG
    // Helping values to make debugging what is happening in a instruction easier
    std::string d_opcode_;
    uint32_t d_length_;
    uint32_t d_result_id_;
    uint32_t d_type_id_;
    // helps people new to using SPIR-V spec to understand Word()
    uint32_t d_words_[12];
#endif
```


LUNARG

Not creating our own IR

- We don’t want to maintain an entire compiler stack
- Have yet to find a true need to make one

Not creating our own IR

- We don’t want to maintain an entire compiler stack
- Have yet to find a true need to make one

Spencer IR

Not creating our own IR

- We don’t want to maintain an entire compiler stack
- Have yet to find a true need to make one

Spencer IR

The Module class

- Created when we get the original SPIR-V
- Stores the instructions in lists
- Each pass updates the Module object

The Module class

- Top level we have InstructionLists and FunctionList

LUNARG

The Module class

- Function have lists of blocks

LUNARG

The Module class

- Function have lists of blocks

LUNARG

The Module class

- Simple way to print out back to SPIR-V

```
for (const auto& inst : annotations_) {
    inst->ToBinary(out);
}
for (const auto& inst : types_values_constants_) {
    inst->ToBinary(out);
}
for (const auto& function : functions_) {
    function->ToBinary(out);
}
```

```
void Instruction::ToBinary(std::vector<uint32_t>& out) {
    for (auto word : words_) {
        out.push_back(word);
    }
}
```

Saying “no” to a global hashmap of IDs

```
if (insn->Opcode() == spv::OpVariable) {
    insn = FindDef(insn->TypeId());
} else if (insn->Opcode() == spv::OpUntypedVariableKHR
    insn = FindDef(insn->Word(4));
} else if (insn->Opcode() == spv::OpTypePointer) {
    insn = FindDef(insn->Word(3));
} else if (insn->IsArray()) {
    insn = FindDef(insn->Word(2));
```

Saying “no” to a global hashmap of IDs

```
if (insn->Opcode() == spv::OpVariable) {
    insn = FindDef(insn->TypeId());
} else if (insn->Opcode() == spv::OpUntypedVariableKHR
    insn = FindDef(insn->Word(4));
} else if (insn->Opcode() == spv::OpTypePointer) {
    insn = FindDef(insn->Word(3));
} else if (insn->IsArray()) {
    insn = FindDef(insn->Word(2));
```

The result can be “any” instruction… why?

LUNARG

Only 5 things you ever need to lookup in SSA

- Metadata (ex. Decorations)
- Instructions in the Function
- Variables
- Types
- Constants

Only 5 things you ever need to lookup in SSA

- Metadata (ex. Decorations)
- Instructions in the Function
- Variables
- Types
- Constants

Only 5 things you ever need to lookup in SSA

- Metadata (ex. Decorations)

```
// InstructionList annotations_;
for (const auto& annotation : annotations_) {
    if (annotation->Opcode() == spv::OpDecorate &&
        annotation->Word(2) == spv::DecorationArrayStride) {
        array_stride = annotation->Word(1);
    }
}
```

Only 5 things you ever need to lookup in SSA

- Metadata (ex. Decorations)
- Instructions in the Function
- Variables
- Types
- Constants

Only 5 things you ever need to lookup in SSA

- Metadata (ex. Decorations)
- Instructions in the Function
- Variables
- Types
- Constants

TypeManger class

- Holds memory for 3 helper structs
  - Type, Variable, and Constant

TypeManger class

- Holds memory for 3 helper structs
  - Type, Variable, and Constant
- Find, Get, and Create functions
  - Find will return of pointer
  - Get will create if not found
  - Create just creates without checking

Keeping the Type object simple

- Really easy to complicate the Types
  - Templated for each type
- We try to not duplicate types
  - … but the world won't burn if there are few duplicated

Control Flow... or lack of

- Trying to really keep things less complex when possible
- Do all our logic in a function call
  - The driver seems to just inline and handle Control Flow for us!
- Compromise: Can't currently validate in a loop header

Hooking up passes

- We do “passes” like most compiler tools
- Each pass is a separate validation check

```
if (gpuav_settings.shader_instrumentation.buffer_device_address) {
    spirv::BufferDeviceAddressPass pass(module);
    modified |= pass.Run();
}

if (gpuav_settings.shader_instrumentation.mesh_shading) {
    spirv::MeshShading pass(module);
    modified |= pass.Run();
}
```

```
if (gpuav_settings.shader_instrumentation.buffer_device_address) {
    spirv::BufferDeviceAddressPass pass(module);
    modified |= pass.Run();
}

if (gpuav_settings.shader_instrumentation.mesh_shading) {
    spirv::MeshShading pass(module);
    modified |= pass.Run();
}
```

```
if (gpuav_settings.shader_instrumentation.buffer_device_address) {
    spirv::BufferDeviceAddressPass pass(module);
    modified |= pass.Run();
}

if (gpuav_settings.shader_instrumentation.mesh_shading) {
    spirv::MeshShading pass(module);
    modified |= pass.Run();
}
```

```
if (gpuav_settings.shader_instrumentation.buffer_device_address) {
    spirv::BufferDeviceAddressPass pass(module);
    modified |= pass.Run();
}

if (gpuav_settings.shader_instrumentation.mesh_shading) {
    spirv::MeshShading pass(module);
    modified |= pass.Run();
}
```

Time to actually validate things!

Adding a validation check

- Always OpFunctionCall to invoke the check
- Isolate logic into the function

Adding a validation check

- Always OpFunctionCall to invoke the check
- Isolate logic into the function
- Process included
  - Writing things in GLSL
  - Seeing the generated SPIR-V
  - Rewrite the C++ to do the same

Adding a validation check

- Always OpFunctionCall to invoke the check
- Isolate logic into the function
- Process included
  - Writing things in GLSL
  - Seeing the generated SPIR-V
  - Rewrite the C++ to do the same

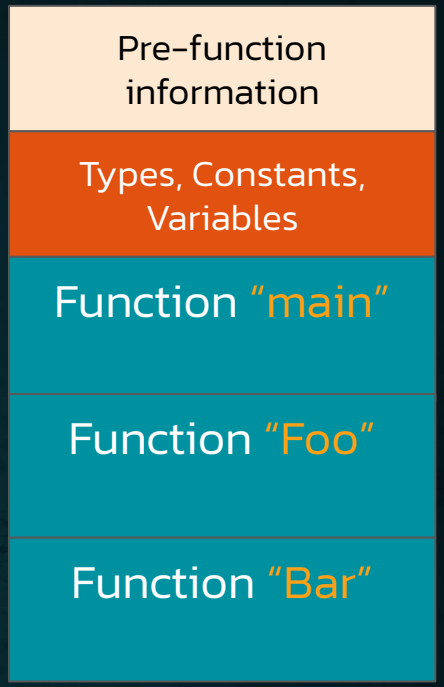
Very slow to iterate!

Adding a validation check

- Always OpFunctionCall to invoke the check
- Isolate logic into the function
- Process included
  - Writing things in GLSL
  - ~~Seeing the generated SPIR-V~~
  - ~~Rewrite the C++ to do the same~~

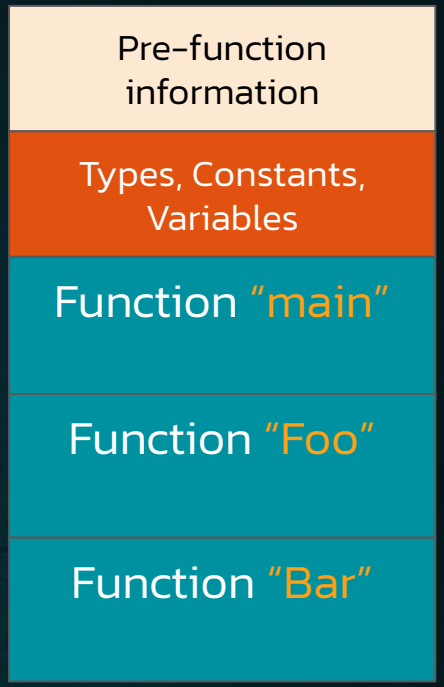
```
vec4 Foo(float x) {
    return vec4(x);
}

vec4 Bar(vec4 x) {
    return x * 2;
}

void main() {
    vec4 a = Foo(1.0);
    vec4 b = Bar(a);
}
```

```
vec4 Foo(float x) {
    return vec4(x);
}

vec4 Bar(vec4 x) {
    return x * 2;
}

void main() {
    vec4 a = Foo(1.0);
    vec4 b = Bar(a);
}
```



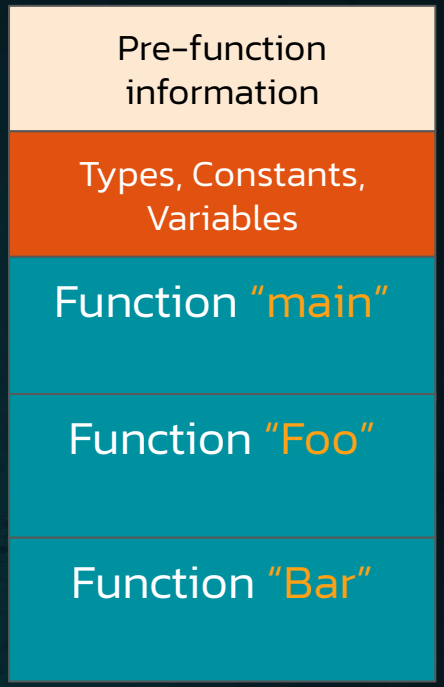
How the app's SPIR-V looks like

LUNARG


Pre-function information
Types, Constants, Variables
Function "main"
Function "Foo"
Function "Bar"

LUNARG

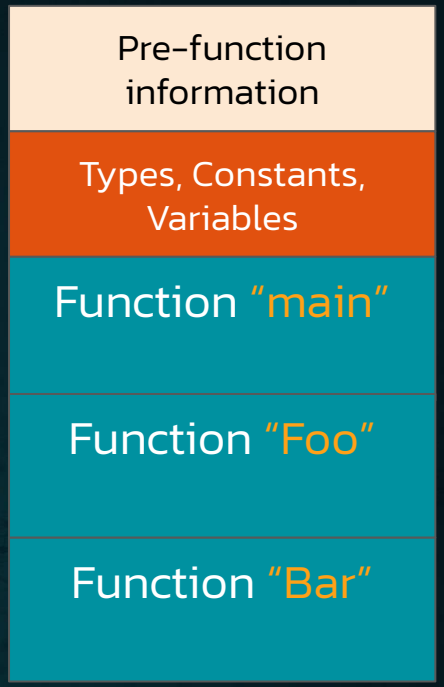
```
#version 450
bool inst_check(uint value) {
    if (value == 0) {
        return true; // error
    }
    return false;
}
```



LUNARG

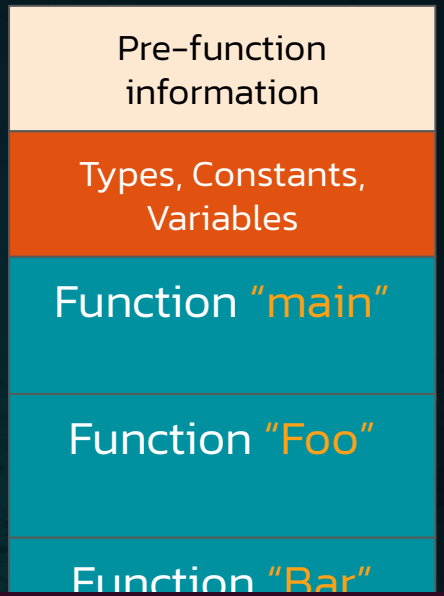
```
#version 450
bool inst_check(uint value) {
    if (value == 0) {
        return true; // error
    }
    return false;
}
```

This is real GLSL code, not pseudo code!



LUNARG

```
#version 450
bool inst_check(uint value) {
    if (value == 0) {
        return true; // error
    }
    return false;
}
```



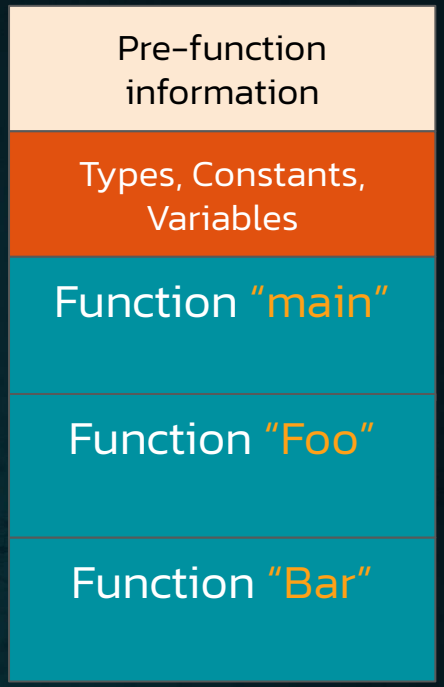
This is real GLSL code, not pseudo code!

```
fricke@fricke:~/sandbox$ glslang -V a.comp -o a.spv
a.comp
ERROR: Linking compute stage: Missing entry point: Each stage requires one entry point

SPIR-V is not generated for failed compile or link
```

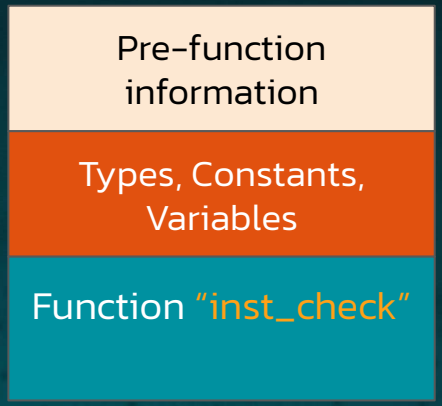

LUNARG

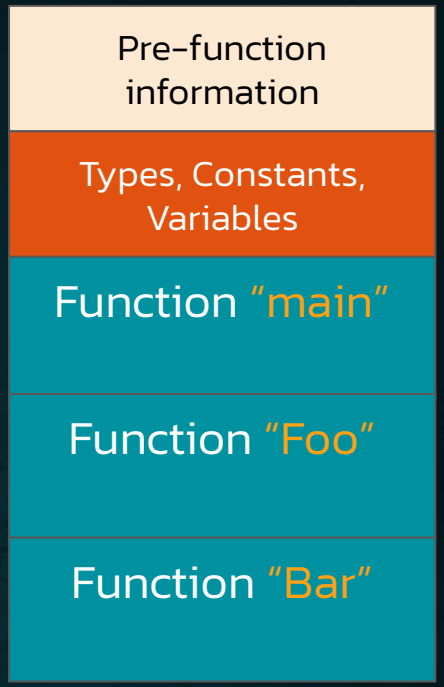
```
#version 450
bool inst_check(uint value) {
    if (value == 0) {
        return true; // error
    }
    return false;
}
```

We created a --no-link option in glslang

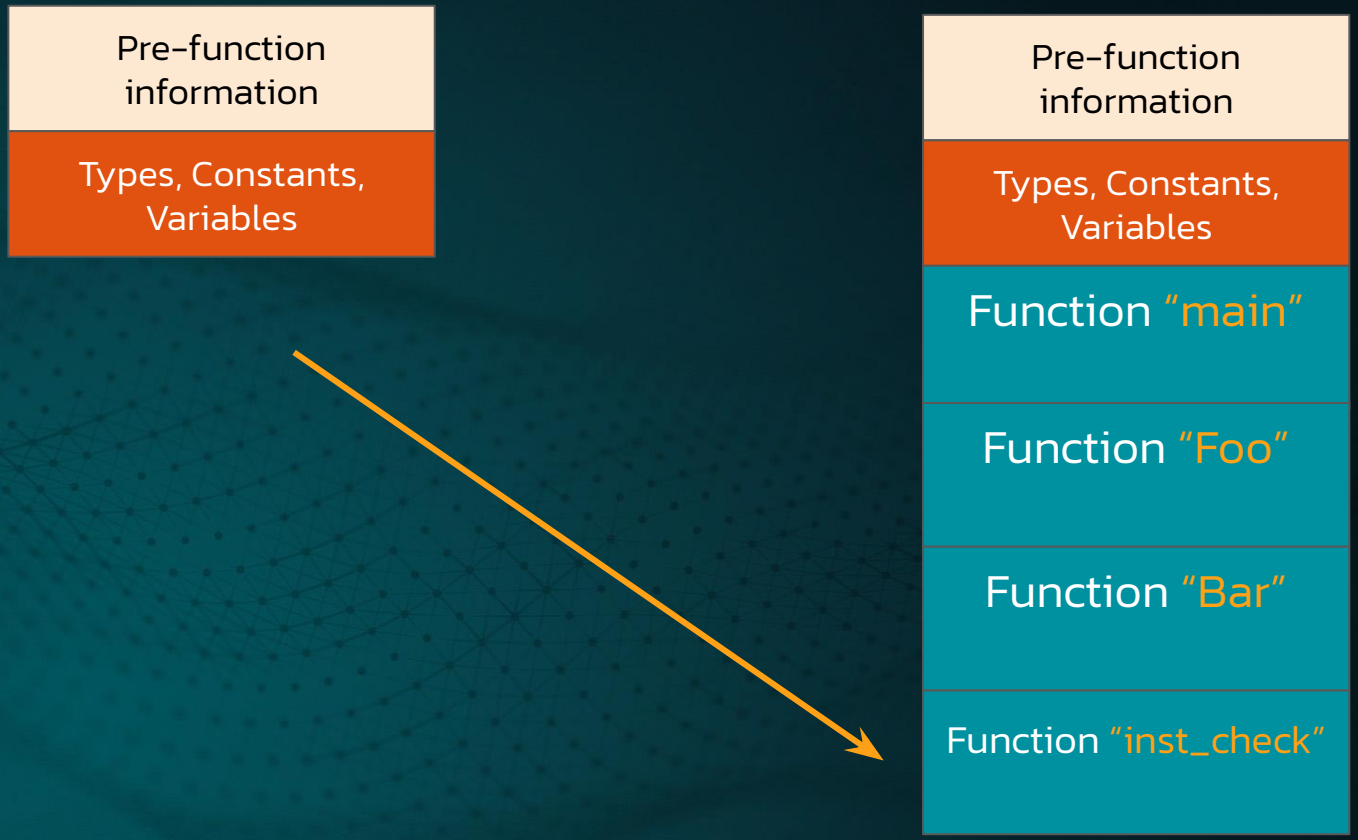


LUNARG

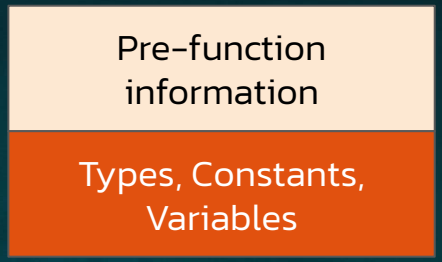

Pre-function information
Types, Constants, Variables
Function "inst_check"

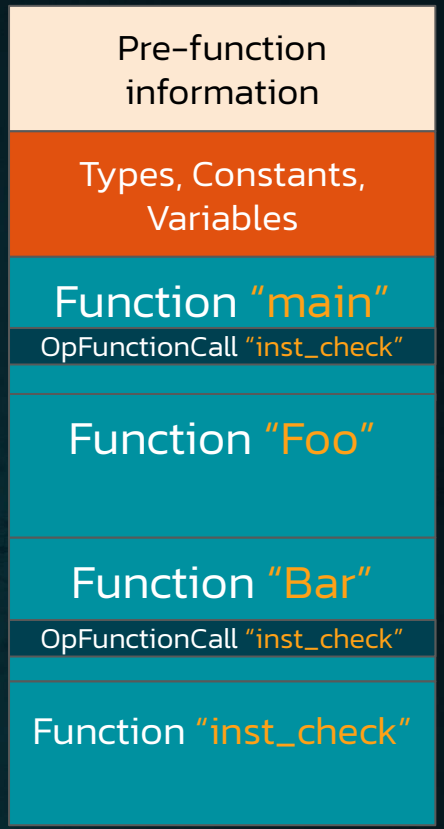

Pre-function information
Types, Constants, Variables
Function "main"
Function "Foo"
Function "Bar"

LUNARG


Pre-function information
Types, Constants, Variables
Pre-function information
Types, Constants, Variables
Function "main"
Function "Foo"
Function "Bar"
Function "inst_check"

LUNARG


Pre-function information
Types, Constants, Variables


Pre-function information
Types, Constants, Variables
Function "main"
OpFunctionCall "inst_check"
Function "Foo"
Function "Bar"
OpFunctionCall "inst_check"
Function "inst_check"

LUNARG

Pre-function information

Add types/constants if don't exist

Pre-function information

Types, Constants, Variables

New Types and Constants

Function "main"

OpFunctionCall "inst_check"

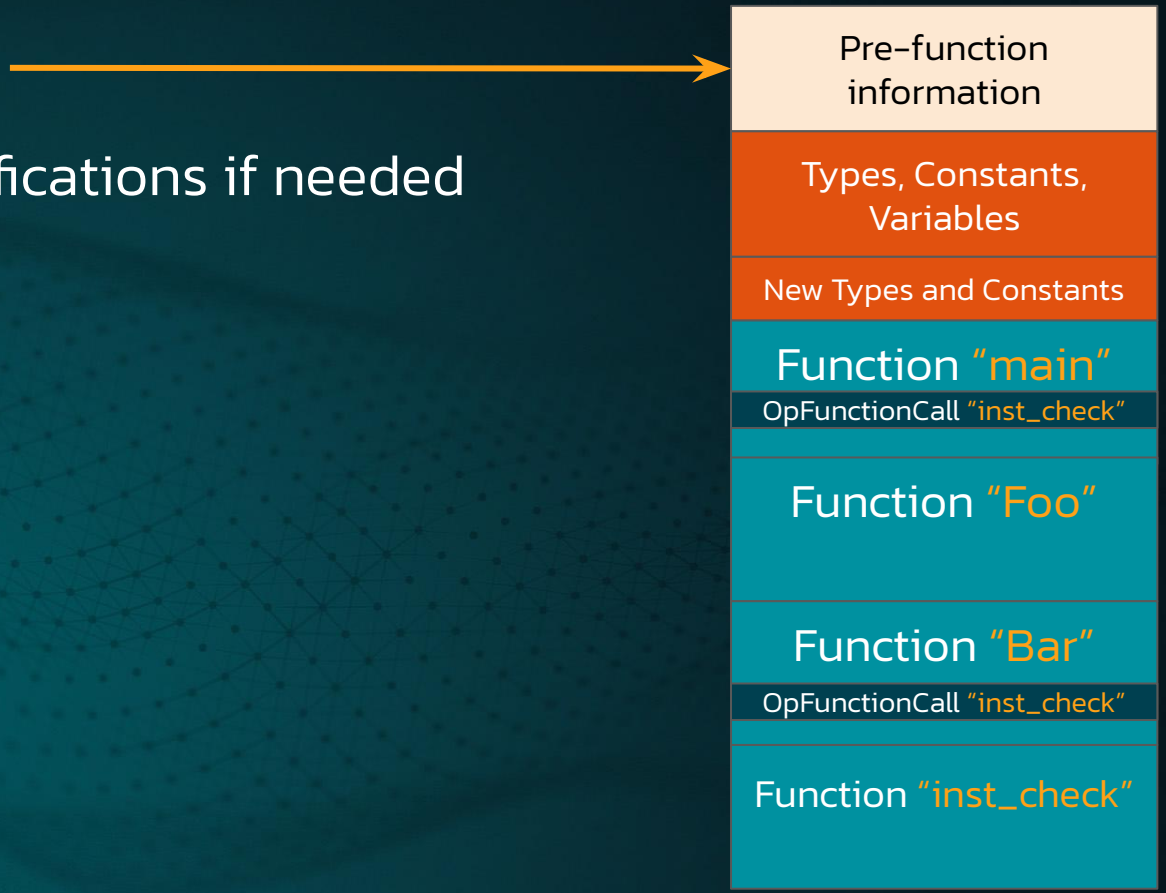
Function "Foo"

Function "Bar"

OpFunctionCall "inst_check"

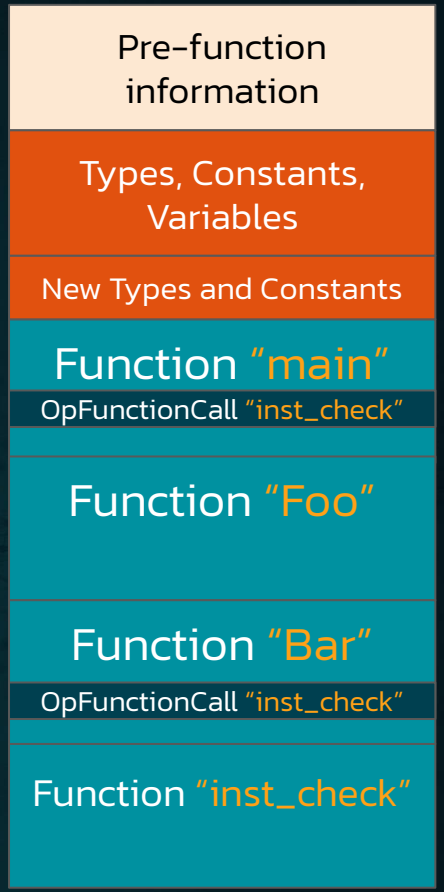
Function "inst_check"

Make any modifications if needed



LUNARG

We can now just write validation in GLSL!



LUNARG

Linker

- Is a single function, not hard to create yourself

Linker

- Is a single function, not hard to create yourself
- Need hashmap of <old id, new id> to swap

Linker

- Is a single function, not hard to create yourself
- Need hashmap of <old id, new id> to swap
- Create new OpType* if can’t find one
  - We leave a few duplicates for complex types like OpTypeStruct

Linker

- Is a single function, not hard to create yourself
- Need hashmap of <old id, new id> to swap
- Create new OpType* if can't find one
  - We leave a few duplicates for complex types like OpTypeStruct
- Add the new functions block and maps to OpFunctionCall

Linker

- Is a single function, not hard to create yourself
- Need hashmap of <old id, new id> to swap
- Create new OpType* if can't find one
  - We leave a few duplicates for complex types like OpTypeStruct
- Add the new functions block and maps to OpFunctionCall
- Only keep the OpName for the function call

Generated operands that need to swap

- The most complex part of linking
- Want to find all the “id” operands
- Used grammar file to find and make a list for each instruction

Generated operands that need to swap

- The most complex part of linking
- Want to find all the “id” operands
- Used grammar file to find and make a list for each instruction

```
{
  "opname" : "OpBranchConditional",
  "class"  : "Control-Flow",
  "opcode" : 250,
  "operands" : [
    { "kind" : "IdRef",                                  "name" : "Condition" },
    { "kind" : "IdRef",                                  "name" : "True Label" },
    { "kind" : "IdRef",                                  "name" : "False Label" },
    { "kind" : "LiteralInteger", "quantifier" : "*", "name" : "Branch weights" }
  ],
  "version": "1.0"
},
```

Generated operands that need to swap

- The most complex part of linking
- Want to find all the “id” operands
- Used grammar file to find and make a list for each instruction

```
{
  "opname" : "OpBranchConditional",
  "class"  : "Control-Flow",
  "opcode" : 250,
  "operands" : [
    { "kind" : "IdRef",                                  "name" : "Condition" },
    { "kind" : "IdRef",                                  "name" : "True Label" },
    { "kind" : "IdRef",                                  "name" : "False Label" },
    { "kind" : "LiteralInteger", "quantifier" : "*", "name" : "Branch weights" }
  ],
  "version": "1.0"
},
```

Don’t want to mess with Control Flow id’s!

Why not use the spirv-link in SPIRV-Tools?

Why not use the spirv-link in SPIRV-Tools?

1. We would require a custom interface

Why not use the spirv-link in SPIRV-Tools?

1. We would require a custom interface
2. Would need to re-parse the SPIR-V again

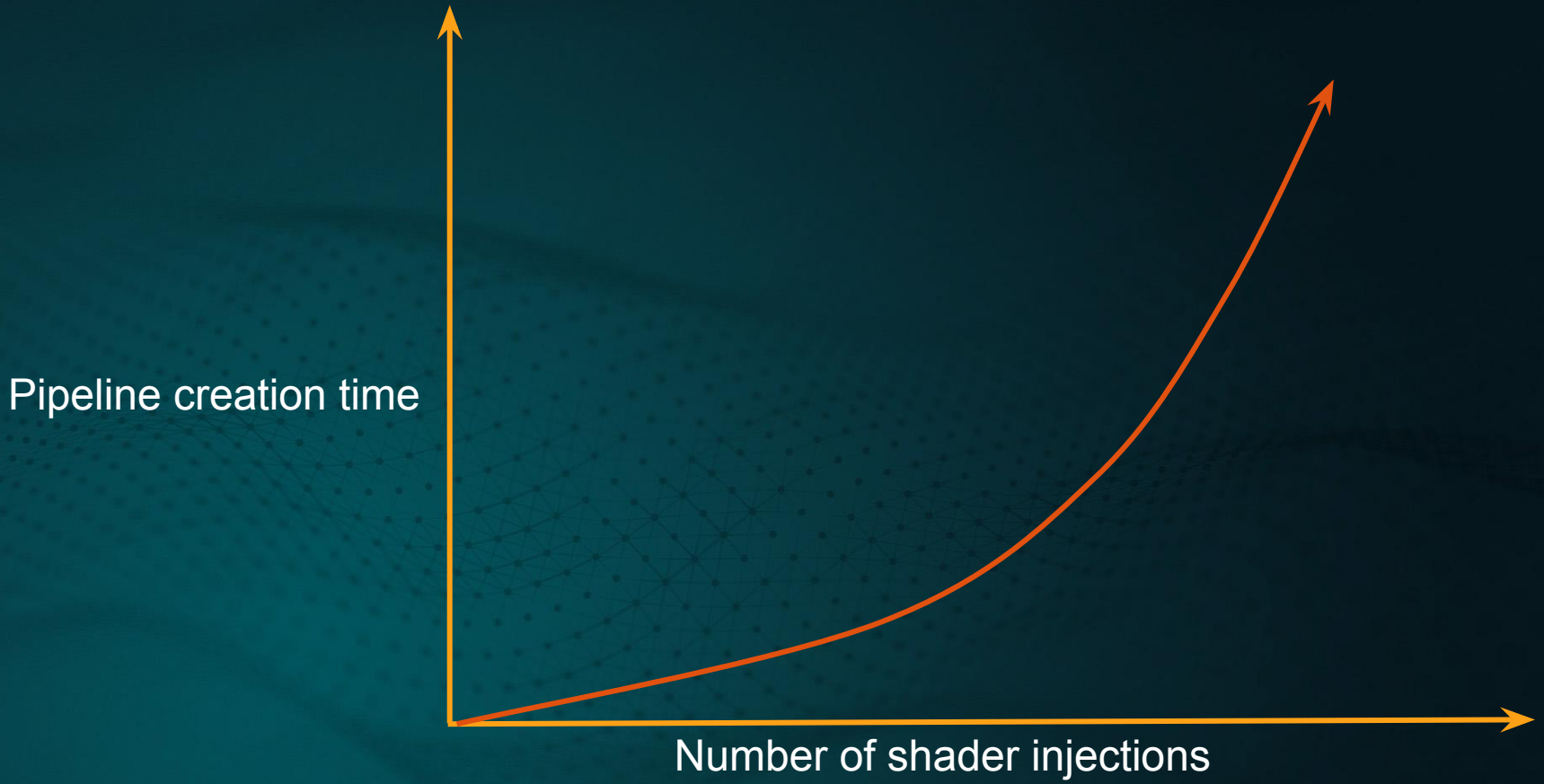
Why not use the spirv-link in SPIRV-Tools?

1. We would require a custom interface
2. Would need to re-parse the SPIR-V again
3. Doing our own linking allows powerful tricks!

Power of linking

- Our GLSL is compiled ahead of time
- We need to embed a runtime handle (uint32_t)
  - Used as a breadcrumbs to report an error
- We use a magic number
  - const uint LinkShaderId = 0x0BAD;
- When linking, search for 0x0BAD, and replace it

So we started adding more and more checks



LUNARG

Real world numbers

- Dota 2 has many shaders needing 50 to 100 injections each
- Normal vkCreatePipeline time – 1ms

Real world numbers

- Dota 2 has many shaders needing 50 to 100 injections each
- Normal vkCreatePipeline time – 1ms
- Adding just a simple function checks – 2ms

Real world numbers

- Dota 2 has many shaders needing 50 to 100 injections each
- Normal vkCreatePipeline time – 1ms
- Adding just a simple function checks – 2ms
- Writing “what happen” to a storage buffer – 15ms

Real world numbers

- Dota 2 has many shaders needing 50 to 100 injections each
- Normal vkCreatePipeline time – 1ms
- Adding just a simple function checks – 2ms
- Writing “what happen” to a storage buffer – 15ms
- Added error logging for each spot – 1000ms

Real world numbers

- Dota 2 has many shaders needing 50 to 100 injections each
- Normal vkCreatePipeline time – 1ms
- Adding just a simple function checks – 2ms
- Writing “what happen” to a storage buffer – 15ms
- Added error logging for each spot – 1000ms
- Adding the actual GPU side validation – 5500ms

Figured out all our complex functions were being inlined!

Open-Source to the rescue

- Can’t overstate the importance of having Mesa drivers
- Was able to actually see how things were being compiled

Everyone who works on Mesa, thanks!

DontInline to the rescue

3.24. Function Control

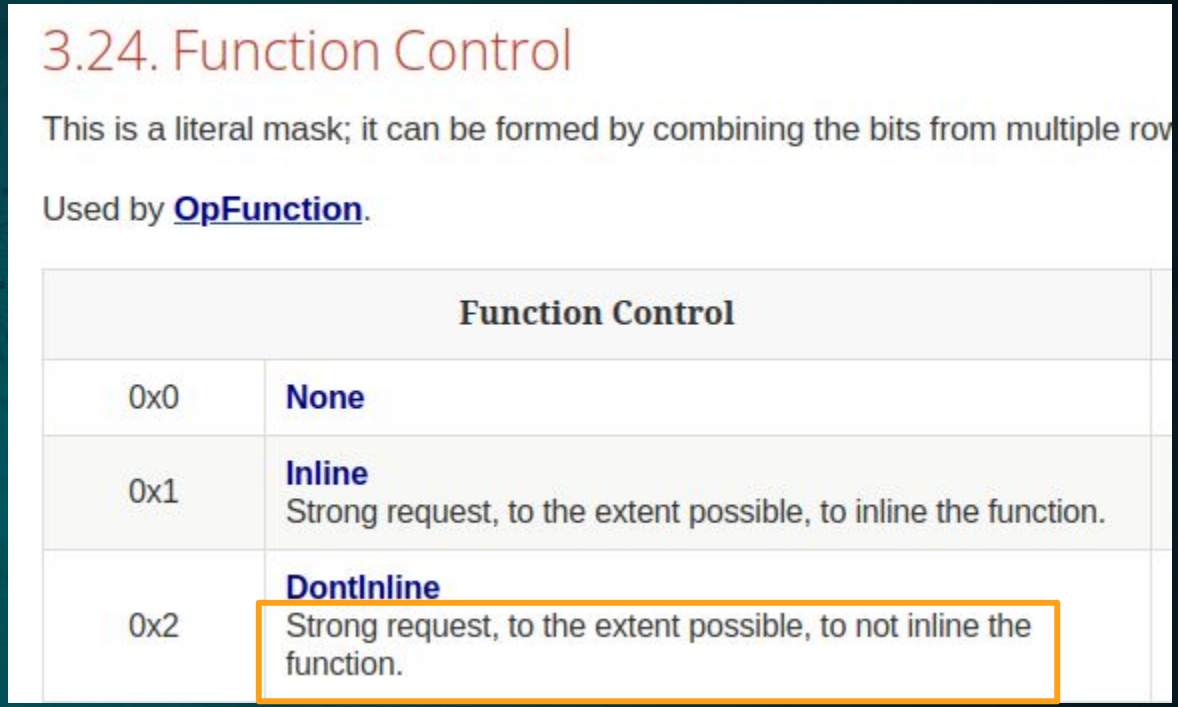
This is a literal mask; it can be formed by combining the bits from multiple rov

Used by **OpFunction**.

Function Control	
0x0	**None**
0x1	**Inline** Strong request, to the extent possible, to inline the function.
0x2	**DontInline** Strong request, to the extent possible, to not inline the function.

LUNARG

DontInline to the rescue



3.24. Function Control

This is a literal mask; it can be formed by combining the bits from multiple rov

Used by **OpFunction**.

Function Control	
0x0	**None**
0x1	**Inline** Strong request, to the extent possible, to inline the function.
0x2	**DontInline** Strong request, to the extent possible, to not inline the function.

LUNARG

DontInline to the rescue

LUNARG

Jeff Bolz (NVIDIA) to the rescue

- Mentioned how they had a flag, just need to connect and test
- 553.31 driver supported it
- 10x speed up instantly!
  - Large apps went from 5 minutes to 25 seconds (on a single thread)

Unfortunately not a solution we could rely on

- No way to query if “supported”
- No way to know if ever honored
- Hard to get every other driver to try and support it

GPU-AV has 2 types of user

1. Those who are debugging a device lost

2. Those who want to just ensure things are “validation clean”

GPU-AV has 2 types of user

Desperate for help, can handle a bit slower workflow

1. Those who are debugging a device lost

2. Those who want to just ensure things are “validation clean”

GPU-AV has 2 types of user

Desperate for help, can handle a bit slower workflow

1. Those who are debugging a device lost

2. Those who want to just ensure things are “validation clean”

Knows the app works, want to make sure it’s
not “by luck” of undefined behavior

Safe Mode

- Have “safe” and “unsafe” mode

Safe Mode

- Have “safe” and “unsafe” mode
- “safe” mode will try to get the vkQueueSubmit to completion
  - Discard bad OpStore
  - Load a zero to OpLoad

Safe Mode

- Have “safe” and “unsafe” mode
- “safe” mode will try to get the vkQueueSubmit to completion
  - Discard bad OpStore
  - Load a zero to OpLoad
- “unsafe” mode will exploit any chance to be faster
  - Have robustness turned on? Will skip those checks
  - Can just store info and validate afterwards on the CPU

2 Ways to inject logic

```
bda.data = 3; // inject here
x = bda.data; // inject here
```

LUNARG

2 Ways to inject logic

```
bda.data = 3; // inject here
x = bda.data; // inject here
```

Unsafe Mode

```
GpuCheck();
bda.data = 3; // inject here

GpuCheck();
x = bda.data; // inject here
```

LUNAR G

2 Ways to inject logic

```
bda.data = 3; // inject here
x = bda.data; // inject here
```

Unsafe Mode

```
GpuCheck();
bda.data = 3; // inject here

GpuCheck();
x = bda.data; // inject here
```

Safe Mode

```
if (!GpuCheck()) {
    bda.data = 3; // inject here
}

if (!GpuCheck()) {
    x = bda.data; // inject here
} else {
    x = 0; // safe value
}
```

Rethinking the problem

- Do you need more than 1 error reported in a shader?
- Majority of time there are no errors to report
- The error logging logic is slow to compile!

Compromise: only report the last error found

Using private variables

```
// Would love to get down to 16-bytes!
struct ErrorPayload {
    uint inst_offset; // to get shader source
    uint shader_error_encoding; // which error
    // data about the error (would love to have more data!)
    uint parameter_0;
    uint parameter_1;
    uint parameter_2;
} error_payload;
```

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                            inst_offset,
                            LinkShaderId |
                                (ErrorGroup << ErrorGroupShift) |
                                (ErrorSubCode << ErrorSubCodeShift),
                            data,
                            data,
                            data);
        return false;
    }
    return true;
}
```

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                            inst_offset,
                            LinkShaderId |
                                (ErrorGroup << ErrorGroupShift) |
                                (ErrorSubCode << ErrorSubCodeShift),
                            data,
                            data,
                            data);
        return false;
    }
    return true;
}
```

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                                inst_offset,
                                LinkShaderId |
                                    (ErrorGroup << ErrorGroupShift) |
                                    (ErrorSubCode << ErrorSubCodeShift),
                                data,
                                data,
                                data);
        return false;
    }
    return true;
}
```

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                            inst_offset,
                            LinkShaderId |
                                (ErrorGroup << ErrorGroupShift) |
                                (ErrorSubCode << ErrorSubCodeShift),
                            data,
                            data,
                            data);
        return false;
    }
    return true;
}
```


LUNARG

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                            inst_offset,
                            LinkShaderId |
                                (ErrorGroup << ErrorGroupShift) |
                                (ErrorSubCode << ErrorSubCodeShift),
                            data,
                            data,
                            data);
        return false;
    }
    return true;
}
```

```
bool inst_validate(uint is_valid, uint inst_offset, uint data) {
    if (!is_valid) {
        error_payload = ErrorPayload(
                            inst_offset,
                            LinkShaderId |
                                (ErrorGroup << ErrorGroupShift) |
                                (ErrorSubCode << ErrorSubCodeShift),
                            data,
                            data,
                            data);
        return false;
    }
    return true;
}
```

But we still have too many instrumented functions....

But we still have too many instrumented functions….

… so let’s just remove them!

Detecting redundant checks

- Only possible with “unsafe” mode
- In a given block
  - hash what we instrumented
  - skip if detected already

Detect ranges for BDA

```
layout(buffer_reference, std430) buffer BDA {
    uint a;
    uint b;
    uint c;
    uint d;
};

BDA ptr;
uint x = ptr.b + ptr.d;
```

We were checking 2 spots here

LUNARG

Detect ranges for BDA

```
layout(buffer_reference, std430) buffer BDA {
    uint a; // [0, 3]
    uint b; // [4, 7]
    uint c; // [8, 11]
    uint d; // [12, 15]
};

BDA ptr;
uint x = ptr.b + ptr.d;
```

Detect ranges for BDA

```
layout(buffer_reference, std430) buffer BDA {
    uint a; // [0, 3]
    uint b; // [4, 7]
    uint c; // [8, 11]
    uint d; // [12, 15]
};

BDA ptr;
uint x = ptr.b + ptr.d;
```

Statically can detect range accessed.
Becomes only 1 check!

LUNARG

Post Processing

- It is fast to just write data to a buffer
- For VK_EXT_descriptor_indeixng
  - We can mark which descriptors were actually accessed
  - Validate on the CPU afterwards

Post Processing

- Way faster to compile the shader
- Way faster to execute on the GPU
- Can write validation logic on in C++ (not GLSL)

Post Processing

- Way faster to compile the shader
- Way faster to execute on the GPU
- Can write validation logic on in C++ (not GLSL)

LUNARG

Post Processing

- Way faster to compile the shader
- Way faster to execute on the GPU
- Can write validation logic on in C++ (not GLS
- Will block presentation thread if slow
- Will not prevent crashing on the GPU

LUNARG

Me everytime I try something new in GPU-AV

before

after

LUNARG

Now things compile faster, runtime is slow!

Flatting the descriptor index

- Vulkan descriptors have 3 level of hierarchy
  - Sets, bindings, index into an array

Flatting the descriptor index

- Vulkan descriptors have 3 level of hierarchy
  - Sets, bindings, index into an array

```
bool validate(uint set, uint binding, uint index) {
    bool is_valid = internal_buffer[set][binding][index];
```

- Expensive to do “look up” on GPU
- Wanted to just embed [set][binding][index] at instrumentation

LUNARG

Realizing vkCreateShaderModule is a lie!

- vkCreateShaderModule isn't the correct spot to instrument
- Need the VkDescriptorSetLayout interface information

Realizing vkCreateShaderModule is a lie!

- vkCreateShaderModule isn't the correct spot to instrument
- Need the VkDescriptorSetLayout interface information
- Moved instrumentation to Pipeline/ShaderObject creation time

Realizing vkCreateShaderModule is a lie!

- vkCreateShaderModule isn't the correct spot to instrument
- Need the VkDescriptorSetLayout interface information
- Moved instrumentation to Pipeline/ShaderObject creation time

Gained speed up embedding descriptor indexing information

LUNARG

But, now lost a standalone executable

- Before could run each instrumentation pass,
  - similar to spirv-opt passes

But, now lost a standalone executable

- Before could run each instrumentation pass,
  - similar to spirv-opt passes
- With VkDescriptorSetLayout, we need the whole pipeline

But, now lost a standalone executable

- Before could run each instrumentation pass,
  - similar to spirv-opt passes
- With VkDescriptorSetLayout, we need the whole pipeline
- Needed to find other ways to quickly test lots of SPIR-V

https://github.com/ValveSoftware/Fossilize

- Layer to capture everything used to create a VkPipleine
- Saves to a .foz file

https://github.com/ValveSoftware/Fossilize

- Layer to capture everything used to create a VkPipleine
- Saves to a .foz file
- fossilize-replay binary to replay pipeline creation
  - Used by Steam to warm up shader cache

https://github.com/ValveSoftware/Fossilize

- Layer to capture everything used to create a VkPipleine
- Saves to a .foz file
- fossilize-replay binary to replay pipeline creation
  - Used by Steam to warm up shader cache
- VERY portable, will skip if your machine can't build the pipeline
- Our way to test shader instrumentation
  - Performance
  - Doesn't crash/assert

Providing error messages

Squeezing every last byte for logging

- Very careful to not waste memory when logging from the GPU
- Make lots of assumptions to share a single dword

Squeezing every last byte for logging

- Very careful to not waste memory when logging from the GPU
- Make lots of assumptions to share a single dword

- No one support more than 32 Descriptor Set

Squeezing every last byte for logging

- Very careful to not waste memory when logging from the GPU
- Make lots of assumptions to share a single dword

- No one support more than 32 Descriptor Set
- Only need a few bits for which shader stage

Squeezing every last byte for logging

- Very careful to not waste memory when logging from the GPU
- Make lots of assumptions to share a single dword

- No one support more than 32 Descriptor Set
- Only need a few bits for which shader stage
- Only track the first 8k draws in a command buffer

Printing the shader source

- We have a copy of the original SPIR-V
- We log the index of which instruction in the module was the issue
- Translate that into a line number
  - NonSemantic.Shader.DebugInfo.100 or OpLine

Having to get the shader source fast

- We were spending too much time logging the shader source
- The bottleneck was converting SPIR-V binary into helper class
  - std::vector<uint32_t> to std::vector<Instruction>

Having to get the shader source fast

- We were spending too much time logging the shader source
- The bottleneck was converting SPIR-V binary into helper class
  - std::vector<uint32_t> to std::vector<Instruction>
- Do our SPIR-V logging with only a uint32_t array
  - Not fun to debug, but it is much faster

How do we debug GPU-AV?

Various debugging options

- Run spirv-val on instrumented shaders
- Dump shaders before and after instrumentation
- Set a maximum instrumentation count

Printing how much instrumentation is done

Printing how much instrumentation is done

- Dump instrumented shader

Printing how much instrumentation is done

- Dump instrumented shader
- Python script searches for “inst_” prefix in function calls

Printing how much instrumentation is done

- Dump instrumented shader
- Python script searches for “inst_” prefix in function calls
- Quickly get stats how much instrumentation was done for entire app

DebugPrintf our shaders

- DebugPrintf already is a pass we implement in GPU-AV
- Add debugPrintf() into our internal GLSL
- Made some adjustments, but can now use!

What still keeps me up at night

Spec constants

OpSpecConstant %uint 4

These are easy and get you 95% of the way

Spec constants

OpSpecConstant %uint 4

These are easy and get you 95% of the way

What happens when you get dozen of

OpSpecConstantOp

Why we use spirv-opt constant folding pass

```
[12]  %10 = OpConstant %8 1
[13]  %3 = OpConstantComposite %9 1 1 1
[14]  %4 = OpSpecConstant %8 0
[15]  %5 = OpSpecConstant %8 16
[16]  %11 = OpTypeBool
[17]  %12 = OpSpecConstantOp %11 OpIEqual spec(0) 1
[18]  %13 = OpConstant %8 2
[19]  %14 = OpSpecConstantOp %11 OpIEqual spec(0) 2
[20]  %15 = OpSpecConstantOp %11 OpLogicalOr %12 %14
[21]  %16 = OpConstant %8 3
[22]  %17 = OpSpecConstantOp %11 OpIEqual spec(0) 3
[23]  %18 = OpSpecConstantOp %11 OpLogicalOr %15 %17
[24]  %19 = OpTypeInt 32 1
[25]  %20 = OpConstant %19 8
[26]  %21 = OpConstant %19 1
[27]  %22 = OpSpecConstantOp %19 OpSelect %18 8 1
[28]  %23 = OpTypeArray %8 %22
[29]  %24 = OpSpecConstantOp %11 OpIEqual spec(0) 1
[30]  %25 = OpSpecConstantOp %11 OpIEqual spec(0) 2
[31]  %26 = OpSpecConstantOp %11 OpLogicalOr %24 %25
[32]  %27 = OpSpecConstantOp %11 OpIEqual spec(0) 3
[33]  %28 = OpSpecConstantOp %11 OpLogicalOr %26 %27
[34]  %29 = OpSpecConstantOp %8 OpSelect %28 spec(16) 1
[35]  %30 = OpTypeArray %23 %29
```

How to replace a bad opaque handle

- When loading a bad int/float can use OpConstantNull
- How to replace a OpTypeImage or OpTypeAccelerationStructure?
  - Found drivers (except lavapipe) seem to just handle a OpConstantNull
    - Not an ideal situation

Final words/advice/thoughts

- Being able to iterate quickly is important
- Create workflows to debug
- Create a way to test and profile against lots of shaders
- Accept creative compromises

Final words/advice/thoughts

- Being able to iterate quickly is important
- Create workflows to debug
- Create a way to test and profile against lots of shaders
- Accept creative compromises

If you have ideas/thoughts, please share them with me after the talk!!!

Final words/advice/thoughts

- Being able to iterate quickly is important
- Create workflows to debug
- Create a way to test and profile against lots of shaders
- Accept creative compromises

Shout out to my colleague Arno for helping with all of this work!

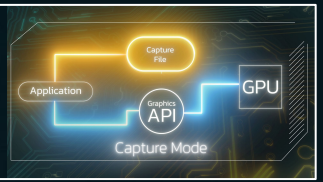
LUNARG
POWER YOUR SUCCESS

Thank you –
KHRONOS GROUP

KosmicKrisp

Come to the LunarG Table!

See KosmicKrisp & GFXReconstruct



Take the 2026 Vulkan Ecosystem Survey!

LunarG Presentations

Vulkanised 2026

LunarG Presentations

Shading Languages Symposium 2026

OpFunctionEnd