SPIRV-Cross

The war stories

• Hans-Kristian Arntzen
• 2018-05-22 – Vulkanise 2018
What this talk is about

• Endgoals and motivation for cross-compilation
• Campfire stories from the trenches with some salt on top
  • GLSL
  • HLSL
  • MSL
• The difficult problems I have faced
  • Freeform rant about shading languages and legacy APIs
• A retrospective on SPIR-V design
• Maybe some practical compiler theory for non-compiler people
  • I started out not knowing anything about common compiler techniques.
Humble beginnings

- September 2015
- Target: GDC 2016 demo
  - Vulkan / GLES comparison showpiece
- SPIR-V 1.0 was basically done
- Glslang was working alright for GLSL -> SPIR-V already
- Needed a more robust shading solution
  - Hacky text-based cross-compiler wasn’t going to cut it for Vulkan GLSL, stretched to its limits.
  - No compute support.
- SPIR-V looked like a viable portable IR-target
  - How hard could it be, right?
The first attempts

Surprisingly, it was not too hard.
GLslang by default emits very simple code.
1:1 match with GLSL syntax tree.
The SPIR-V spec is really nice to read and understand as a standalone document.
Donating to Khronos

Released spir2cross for GDC 2016.
Renamed to SPIRV-Cross and moved to KhronosGroup/ shortly after.
I have continued maintaining and developing it since.
The GLSL skirmish
Clean output by design

Looking at you, hlslccc 😞
Readability is important

Older cross-compilers like hlslcc and cgc emit line noise.
Part of the reason for this is using optimized FXC output as a baseline.
Readability is not important when using GLSL as a «compiled target» that just has to run.
SPIRV-Cross aims for readability

SPIR-V can be very clean, so I really want to make use of that.
A very important use case for our driver developers was using the tool to understand SPIR-V without decrypting SPIR-V asm.
SPIR-V asm is a pretty painful language to read and write, as is any IR-like format.
Readable code matters in debug, modify cycles

```gls
float normal_scale;

layout(set = 2, binding = 0) uniform medium sampler2D uBaseColormap;
layout(set = 2, binding = 1) uniform medium sampler2D uNormalmap;
layout(set = 2, binding = 2) uniform medium sampler2D uMetallicRoughnessmap;
layout(set = 2, binding = 3) uniform medium sampler2D uOcclusionMap;

layout(location = 1) out medium vec4 BaseColor;
layout(location = 2) out medium vec3 Normal;
layout(location = 3) out medium vec2 PBR;
layout(location = 11) in vec2 vUV;
layout(location = 12) in medium vec3 vNormal;
layout(location = 13) in medium vec4 vTangent;
layout(location = 14) in medium vec3 vEyeVec;

void emit_render_target(medium vec3 emissive, medium vec4 base_color, medium vec3 normal, medium float metallic, medium float roughness, medium !

void main()

medium vec4 base_color = texture(uBaseColormap, vUV, registers.lod_bias) * registers.base_color;
medium vec3 normal = normalize(vNormal);
medium vec3 tangent = normalize(vTangent.xzy);
medium vec3 binormal = cross(normal, tangent) * vTangent.w;
medium vec3 tangent_space = (texture(uNormalmap, vUV, registers.lod_bias).xy * 2.0) - vec2(1.0);
tangent float tangent_z = sqrt(max(0.0, 1.0 - dot(tangent_space, tangent_space)));
tangent_space *= registers.normal_scale;
```
Loops

First major hurdle
Loops are critical for performance

Whether or not a loop is unrolled in the right way can have game-changing performance.

Emitting weird loops is also a great way to crash drivers.
  - Compiler frontends, especially GLSL, are only trained on «sane» loops.

At this point, I realized I needed some compiler-fu knowledge.
The loop anatomy

• Initialize i

• %header: OpLoopMerge %merge-block %continue-block
  • %cond = Compare(i < N)
  • SelectionBranch %cond %body %merge-block
  • %body:
    – Do stuff
    – OpBranch %continue-block
• %continue-block:
  – i++
  – OpBranch %header
Naive translation

```cpp
int i = 0;
for (;;){
    if (i < N){
        // body
        i++;
        continue;
    } else {
        break;
    }
}
```

LoopMerge, aka. how to crash and burn a GLSL compiler.
Enter control-flow graph analysis

Construct a CFG.
Find all variables accessed in continue block (where i++ happens).
Find dominating block which accesses i.
  • Fortunately, this algorithm is simple because SPIR-V is structured, YAY! :D
Analyze if there is a direct branch pattern from dominator to loop header.
  • Need to know if i has a statically known expression in the header, like i = 0.
From merge block, verify that there is no access to the variable.
Make sure that the first block in loop does not force temporaries before getting to the loop conditional.
Use common patterns to detect which OpLoopMerges are for loops, while loops, do-while, etc ...
Better translation

```c
for (int i = 0; i < N; i = i + 1)
{
    // body
}

// Look ma’, I can be a compiler nerd too! :D

// Just took 4 months to figure all of this out 😞
ESSL 1.0 == Extremely Stupid Shading Language 1.0

This should just have been a nice optimization, turns out ESSL 1.0 is being really annoying.

- Kinda like HLSL legalization, except for loops ...

Also need to do a text-based optimization to transform:

- \( i = i + 1 \) into
- \( i += 1 \)

for loops are supported but with the following restrictions:

- There is one loop index.
- The loop index has type int or float.
- The for statement has the form:

  \[
  \text{for} \ (\text{init-declaration} \ ; \ \text{condition} \ ; \ \text{expression}) \ \text{statement}
  \]

- init-declaration has the form:

  \[
  \text{type-specifier} \ \text{identifier} = \text{constant-expression}
  \]

Consequently the loop variable cannot be a global variable.

- condition has the form

  \[
  \text{loop_index relational_operator constant_expression}
  \]

  where relational_operator is one of \( \geq \), \( \leq \), \( < \), \( > \) or \( != \)

- for_header has one of the following forms:

  \[
  \text{loop_index}++
  \]

  \[
  \text{loop_index}--
  \]

  \[
  \text{loop_index} += \text{constant_expression}
  \]

  \[
  \text{loop_index} -= \text{constant_expression}
  \]

- Within the body of the loop, the loop index is not statically assigned to nor is it used as the argument to a function or inout parameter.
Phi nodes

COMEFROM is cool, right?
Single Static Assignment is weird

I am sure compiler nerds love it, but it can be awkward to translate to high-level code. Phi is a weird construct to support statically assigned temporaries even with branching logic. Was not going to read a truckload of dry theory to understand why this is useful.
Translate OpPhi back to load-store

Find all OpPhi temporaries, think of them as normal GLSL variables.
  • Declare them at some appropriate scope.

Anatomy of a Phi node:
  • %id OpPhi %type
  • %if-came-from-block-id %become-temporary-id
  • %if-came-from-block-id %become-another-temporary-id

Record commands to do at end of a block:
  • If we branch from block N to M, copy expression X to Phi variable Y.
  • Can happen at any time ... AT ANY TIME
  • Some bugs because not flushing Phi at really awkward moments.

Loop variables can be Phi nodes ...
  • Oh, hello SPIRV-opt
  • Not fun to deal with 😞
Variable scoping
CFG and dominator analysis solves everything, right?

The «correct» place to emit a variable declaration is in its inner scope. Normal logic dictates that the dominating block represents the inner scope.
Inner scope dominating outer scope

In the CFG world, the control flow here is completely linear!
Enter fake branch hack

Let’s invent a branch that doesn’t exist.
For every loop, pretend there exists a direct branch from loop header to merge target.
This forces the dominator to be outside any loop if a variable is accessed outside the loop.

```cpp
int i; // Dominator is now here
do {
    // Pretend that we can branch over this block.
    i = 10;
    break;
} while (false);

FragColor = vec4(i);
```
Continue block dominators

In some really dumb scenarios, continue blocks can become a dominator.
  - Continue blocks cannot declare variables.

They can produce variables which are consumed by outer scope.
Similar hacks, pretend these accesses also happen in loop header to «hoist» out declarations ...

```c
for (int i = 0; i < N; o_hi_I_guess_im_a_dominator_now, i++)
{
}
```
Did I mention temporaries?

For a long time, I only did this analysis on variables.
Easy to track use, OpLoad, OpStore, OpAccessChain, etc ...
Temporaries can be used all over the place.
Temporaries also need full CFG and dominator analysis.
  • Thanks SPIRV-opt! 😊

Phi variables are also temporaries it turns out.
  • Basically need to check everything ...

Fortunately, SPIR-V has a very regular way of encoding instructions
  • Result type
  • Result ID
  • ID arguments ...
  • Few exceptions
Combined image samplers
Separate textures and samplers

All modern APIs have moved to this model. OpenGL is not a modern API (and neither is DX9 for that matter). Need to make SPIR-V from Vulkan GLSL and HLSL work on ancient targets.
The problem

```glsl
uniform texture2D Textures[4]; // OpTypeImage
uniform sampler Sampler;       // OpTypeSampler

// %combined = OpSampledImage %image %sampler
FragColor = texture(sampler2D(Textures[1], Sampler), UV);
```

```
uniform sampler2D Textures[4]; // OpTypeSampledImage
FragColor = texture(Textures[1], UV);
```
Solution

This is not a trivial solution. We need to invent new variables, and hide others. Analyze the entire call-stack, and figure out every possible combination of sampler and image. This analysis must understand function calls. Need to be able to translate function arguments into the callers input, recursively.

Declare all possible combinations we found during analysis.

While emitting code, hijack OpSampledImage to redirect to our combined sampler. User need to call build_combined_image_samplers(), and fixup binding points for the new combined image samplers. These kind of hacks are non-trivial to integrate, but doable.

• New combined sampler ID
• Image ID
• Sampler ID
Buffer flattening

ESSL workarounds are fun
Ancient uniform support

Uniform buffers (constant buffers in HLSL) and push constants are the ways to go these days. Neither of these exist in:

• GLES 2.0 (sorry)
• GL 2
• WebGL 1.0
Obvious solution, enter plain uniforms

```glsl
layout(std140, binding = 0) uniform UBO {
    mat4 MVP;
    vec3 cam_pos;
} ubo;

struct UBO {
    mat4 MVP;
    vec3 cam_pos;
};
uniform UBO ubo;
```

Pre-compile a series of glUniform*() calls to update UBO to members using reflection API.
Even if we could use GLES 3.0 UBOs ...

Old mobile drivers are great, right? 😊

`glUniform4fv` is apparently the way to go for bulk uniform update on GLES.

**Arseny from Roblox contributed buffer flattening support.**

This came in handy later in other backends ... 😊
Flat array

```cpp
layout(std140, binding = 0) uniform UBO {
    mat4 MVP;
    vec3 cam_pos;
} ubo;

uniform vec4 UBO[5];
// Update once with glUniform4fv.

// Flatten all access.
// Relies on 16 byte stride for UBO array access ...
mat4 MVP = mat4(UBO[0], UBO[1], UBO[2], UBO[3]);
vec3 cam_pos = UBO[4].xyz;
```
Shadow samplers
Close to GLSL, but not closer
Do we really need separate sampler types?

```cpp
uniform sampler Sampler;
uniform texture2D Texture;

float pcf = texture(sampler2DShadow(Texture, Sampler), UVRef);
```
Do we really need separate sampler types?

```cpp
uniform samplerShadow Sampler; // GRRRRRRRRRRRRRRR
uniform texture2D Texture;

float pcf = texture(sampler2DShadow(Texture, Sampler), UVRef);
```

HLSL has same issue ...
SamplerState vs SamplerComparisonState
SPIR-V has no concept of shadow samplers

OpTypeSampler has no shadow state

Well, here we go again ...

• Analyze all uses of OpSampledImage in call chain
• Check depth state ...
• Build a set of samplers which have been used as sampler2DShadow and friends ...

Type system now needs to know which variable it is used for

• Translate OpTypeSampler to either sampler or samplerShadow
• Similar for HLSL
Metal

Allied forces join the battle
MoltenVK joins the effort

Bill Hollings from Brenwill contributed a MSL backend and switched over to SPIRV-Cross. Laid the groundwork for how we would support multiple backends going forward.
Metal shading language is weird

Declaring resources is completely different from other languages.

NO GLOBALS
CRAM EVERYTHING INTO MAIN()

Solution? Analyze and propagate used resources down to leaf functions ... Fake arguments.

MSL also has some very funny ideas about how vertex buffers should work.

Overall, MSL is the hardest language to target from cross-compilation.
I actually do not have a machine which can run or compile Metal code.

No open source compiler for MSL 😞

Fortunately, Travis can compile MSL for us 😊
Arrays are pointers, right? 😐

```cpp
template<typename T, uint N>
void spvArrayCopy(thread T (&dst)[N], thread const T (&src)[N])
{
    for (uint i = 0; i < N; dst[i] = src[i], i++);
}

template<typename T, uint N>
void spvArrayCopyConstant(thread T (&dst)[N], constant T (&src)[N])
{
    for (uint i = 0; i < N; dst[i] = src[i], i++);
}
```
Forced constexpr samplers? No, no, noooooooooooooo

iOS metal seems to require compile-time constant sampler for sample_compare #533

Closed cyh36 opened this issue 26 days ago · 8 comments
HLSL joins the club

Robert Konrad contributed an HLSL backend.
Now we had coverage for all major shading languages.

Early days

- Bare bones in terms of features, but the overall structure was there.
- I have improved it over time as I have learned the dark art of HLSL.
Matrix

Should have taken the blue pill ...
Matrix conventions in HLSL are awkward

Default memory layout is **column major**.

Access pattern when indexing a matrix is **by row**.

- Similar with construction ...

SPIR-V, GLSL, MSL are all column major by default.

Glslang inverts the matrix memory layout, and flips any multiplication order.
Steal the trick when cross-compiling to HLSL 😊

// Flip the convention. HLSL is a bit odd in that the memory layout is column major ... but the language API is "row-major".
// The way to deal with this is to multiply everything in inverse order, and reverse the memory layout.

if (flags.get(DecorationColMajor))
    return "row_major ";
else if (flags.get(DecorationRowMajor))
    return "column_major ";
Storage buffers

I’ve got 99 buffer types, but SSBO ain’t one
SSBO is apparently hard to support for HLSL

HLSL has three major types of UAVs
  • StructuredBuffer
  • ByteAddressBuffer
  • Append/ConsumeBuffer

SSBOs are more explicit
  • Normal members
  • Last member can be an unsized array, like C99

StructuredBuffer is clean, but ...
  • Assumes the buffer block only contains a single unsized array
  • Impossible to express flexible stuff like SSBO and SPIR-V can 😊

ByteAddressBuffer it is ...
  • Flatten everything, ESSL flattening workarounds reuse 😊
  • Use Load/Store, Load2, Load3, Load4 and build up types ... Bitcast from uint to appropriate type.

```cpp
buffer SSBO {
  int a;
  int b[4];
  vec4 unsized_array[];
};
```
Load row-major matrix from SSBO?

```glsl
#version 450
layout(location = 0) in vec4 position;
lazy(binding = 0, std430, row_major) readonly buffer SSBO { mat4 MVP; }
void main()
{
    gl_Position = MVP * position;
}

void vert_main()
{
    float4x4 _22 = asfloat(uint4x4(
    gl_Position = mul(position, _22);
}
```
SPIRV-opt
Because life has to be difficult
SPIRV-opt is a conspiracy to break SPIRV-Cross

In the early days, SPIRV-opt did not do much interesting.
Very little or no test coverage for «interesting» SPIR-V with awkward control flow in SPIRV-Cross.
Over the last year or so, things have changed. SPIRV-opt does useful things now 😊, albeit in strange, non-intuitive ways 😞
Countless bugs have been filed for silly things. I’ve been swatting flies ever since.
Seems to have stabilized now.
The test suite in SPIRV-Cross tests almost all shaders with SPIRV-opt roundtrip as well.
Lesson learned:
«Good» code in IR can be really stupid in high-level code.
Continue blocks are not just for incrementing i

SPIRV-opt had the idea of simplifying for loops.
The continue block is the only block in SPIR-V which can loop back to the header.
If there is no control flow, why not put ALL THE THINGS in the continue block?
Continue blocks are not just for incrementing \( i \)

SPIRV-opt had the idea of simplifying for loops.
The continue block is the only block in SPIR-V which can loop back to the header.
If there is no control flow, why not put ALL THE THINGS in the continue block?

```c
for (int i = 0; i < 10; put, all, the, code, on, one, line, because, why, not, this, is, totally, legal, but, so, is, being, $PROFANITY, i++)
{
}
```

Workaround? Detect this pattern and move continue block into the body instead ...
Phi can be a serious pain

In raw glslang output, Phi pops up once in a while.

Meet SPIRV-opt!

• OpPhi takes other OpPhi variables as input
• OpPhi ordering is important because OpPhi depends on temporary being latched in current block.
• Inventing fake loop variables because OpPhi all the things.

Bad GLSL code generated from optimized SPIR-V #494

Closed  turol opened this issue on Mar 8 · 5 comments

turol commented on Mar 8

SPIRV-Cross generates GLSL code with different behavior from two shaders which should be functionally equivalent. The only difference between these shaders is that the bad version has been run through the optimizer. When testing with Vulkan where SPIRV-cross is not used the shaders behave identically. Currently
Avoiding total annihilation by inlining

Inlining can be surprisingly painful.

C-like languages have one escape hatch to structured control-flow:
  • return

A structured workaround for this is wrapping the inlined function in do/while with break.
  • Does not work if returning from within another loop however.

The inline pass in SPIRV-opt exposed countless issues with variable and temporary scope.
#version 450
layout(location = 0) out float FragColor;
float inlined_function(vec2 uv)
{
    if (uv.x < 0.4)
        return uv.y;
    if (uv.y < 0.6)
        return uv.x;
    return uv.x + uv.y;
}

void main()
{
    FragColor = inlined_function(gl_FragCoord.xy);
}

#version 450
layout(location = 0) out float FragColor;
float inlined_function(vec2 uv)
{
    if (uv.x < 0.4000000059604644775390625)
    {
        return uv.y;
    }
    if (uv.y < 0.60000002384185791015625)
    {
        return uv.x;
    }
    return uv.x + uv.y;
}

void main()
{
    vec2 param = gl_FragCoord.xy;
    FragColor = inlined_function(param);
}
void main()
{
  bool _70 = gl_FragCoord.x < 0.4000000059604644775390625;
  float _110;
  if (_70)
  {
    _110 = gl_FragCoord.y;
  } else
  {
    _110 = _111;
  }
  bool _114 = _70 ? true : false;
  float _109;
  if (!_114)
  {
    bool _80 = gl_FragCoord.y < 0.60000002384185791015625;
    float _115 = _80 ? gl_FragCoord.x : _110;
    float _112;
    if (!(_80 ? true : _114))
    {
      _112 = gl_FragCoord.x + gl_FragCoord.y;
    } else
    {
      _112 = _115;
    }
    _109 = _112;
  } else
  {
    _109 = _110;
  }
  FragColor = _109;
}
Buffer packing

Level 99 dragons be here
Obligatory XKCD

How Standards Proliferate:
(See: A/C chargers, character encodings, instant messaging, etc.)

Situation: There are 14 competing standards.

14?! Ridiculous! We need to develop one universal standard that covers everyone’s use cases. Yeah!

Soon:

Situation: There are 15 competing standards.
Buffer packing is a complete nightmare

SPIR-V has explicit offsets, array strides and matrix strides for everything.
GLSL, HLSL, MSL are all severely lacking in some way.
Cannot express directly what SPIR-V uses.
No perfect solution, can only approximate, and gracefully fail.
Legacy restrictions continue to pollute our addressing models.
No way to fallback to reinterpret_cast<T*>(ptr + addr).
Let us look at some standards

vec3 is the root of all evil.

GLSL std140 / std430:
• V3 has offset 0, size 12
• V1 has offset 12, size 4

HLSL cbuffer:
• V3 has offset 0, size 12
• V1 has offset 12, size 4

MSL:
• V3 has offset 0, size 16
• V1 has offset 16, size 4

```cpp
buffer {
    vec3 v3;
    float v1;
};
```
The suffering does not end here

GLSL std140 / std430:
  • V1 has offset 0, size 4
  • V3 has offset 16, size 12

HLSL cbuffer:
  • V1 has offset 0, size 4
  • V3 has offset 4, size 12 (stupid float4 straddle boundary rule ...)

MSL:
  • V1 has offset 0, size 4
  • V3 has offset 16, size 16
Fun with arrays

GLSL std140:
- Array stride 16

GLSL std430:
- Array stride 4

HLSL cbuffer:
- Array stride 16

MSL:
- Array stride 4 or 16 (not sure, to be honest :D)
Mitigation strategies

SPIR-V can do whatever it wants, so we need to do our best to map the SPIR-V onto broken shading languages.

Options we have to deal with it:

- layout(offset = N) in modern GLSL (forget about older targets)
- packoffset(c#.swiz) in HLSL cbuffer (Does not work with ConstantBuffer<T>! 😞)
- Rewrite buffer members to realign (only solution for MSL)

Notice, no way to control array stride or matrix stride.

layout() and packoffset only applies to top-level buffer block, not child structs.

Flattening everything ...

- Absolutely disgusting code
- Only option for ByteAddressBuffer in HLSL
- Need to alias blocks for float/vec2/vec3/vec4, maybe viable in GLSL
Conclusion
SPIR-V design

The fact that SPIRV-Cross even worked out in the end can be attributed to SPIR-V.

It is structured:

- Can reliably translate SPIR-V to high-level concepts without a GOTO soup.
- SelectionMerges and LoopMerges are critical to understand the control flow without extensive analysis.

Higher level types are supported:

- Structs and matrices
- This is critical to be able to emit code which is readable (oh hi, hlsllcc).
- Reflection friendly, doesn’t have to emit a side-channel of reflection data (FXC?)

Support for Load/Store model:

- Very important for initial bringup
- Would probably have dropped it if I had to support full SSA shenanigans from day 1.

SPIR-V is future proof:

- Not tied to a particular LLVM version (oh hi, DXIL).
State of SPIRV-Cross development

Ready for «serious» use.
- Almost all bugs now tend to be rather trivial in nature and are fixed rapidly.
- Ships in MoltenVK, DOTA2 on Metal, part of Vulkan portability initiative, used by various open source projects.
- Used by Roblox for their Metal backend.
- I want SPIR-V to be the one true shading language IR for the future. SPIRV-Cross will keep legacy running.

Over the last year or so, I have been more reactive than proactive.
- Fixing bugs and issues as they come up.
- Dealing with PRs, normal day-to-day maintainer stuff.

I can only spend so much time on SPIRV-Cross.
- I prioritize fixing bugs over new features.
- I have managed to keep turnaround time very low, usually a few days for bug filed to committed PR.
- Feature requests take a while unless trivial.

Contributions welcome, big and small
- We have some regular contributors, thank you :D
- Want to encourage new features to mostly come from contributors.
KhronosGroup / SPIRV-Cross

22 Open, 245 Closed

1. Uber-reflection format [enhancement, help wanted, question]
   #544 opened 16 days ago by HansKristian-ARM

2. Question about half / mediump [question]
   #527 opened on Apr 8 by alek314

3. Support 16-bit integer types. [enhancement]
   #491 opened on Mar 7 by HansKristian-ARM

4. Typify uint32_t in cpp public API [enhancement]
   #441 opened on Feb 8 by Sairony

5. Refactor old loop_dominator tracking. [enhancement]
   #423 opened on Feb 1 by HansKristian-ARM
Pull requests do not linger for long
Still going strong

Feb 28, 2016 – May 9, 2018

Contributions to master, excluding merge commits

Contributions: Commits ▼
Non-trivial missing features

HLSL
- Tessellation
- Geometry shaders

MSL
- Tessellation
- Fortunately, no geometry shaders in Metal 😊
Thank You
Danke
Merci
谢谢
ありがとうございました
Gracias
Kiitos
감사합니다
धन्यवाद
תודה