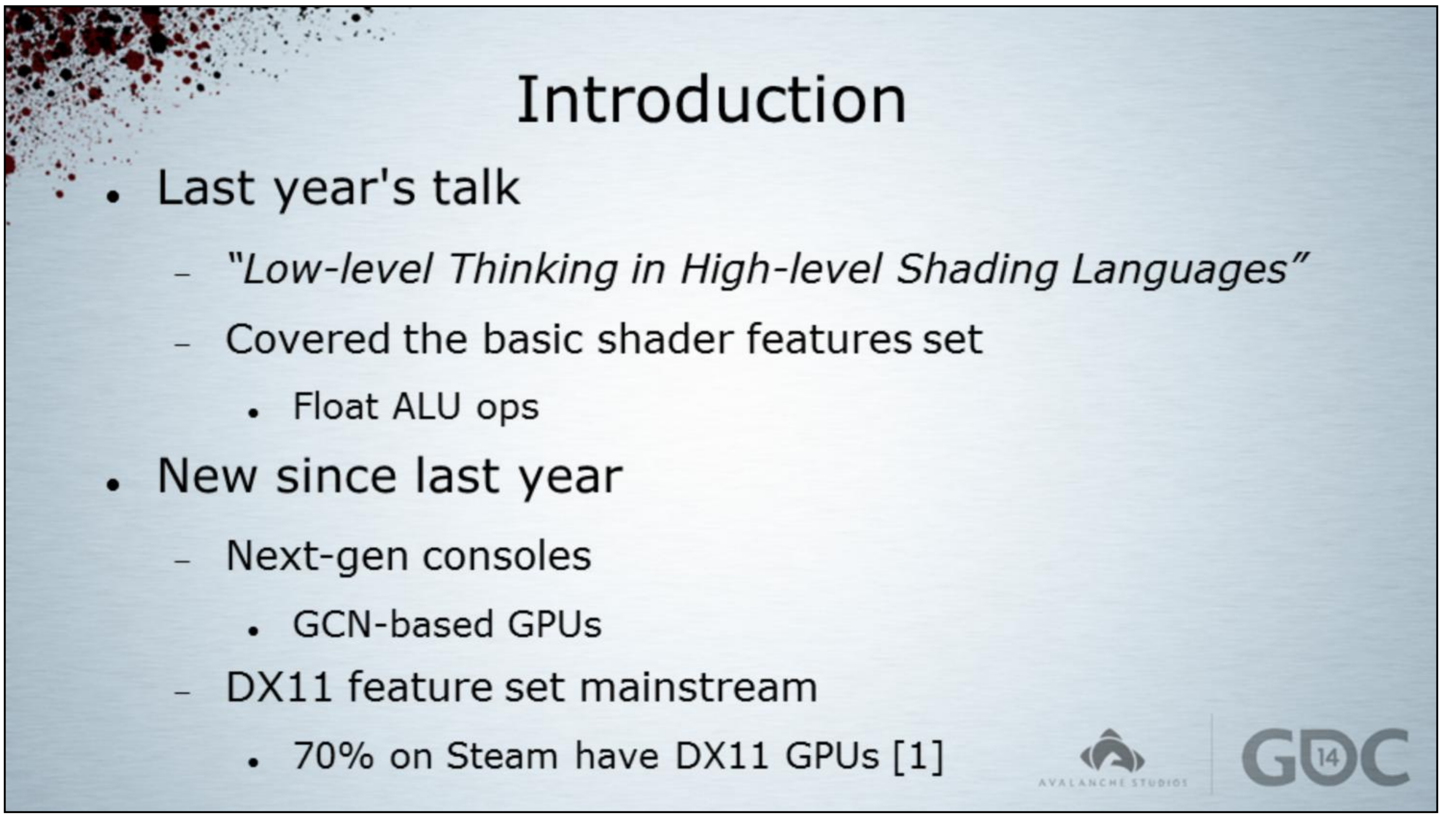
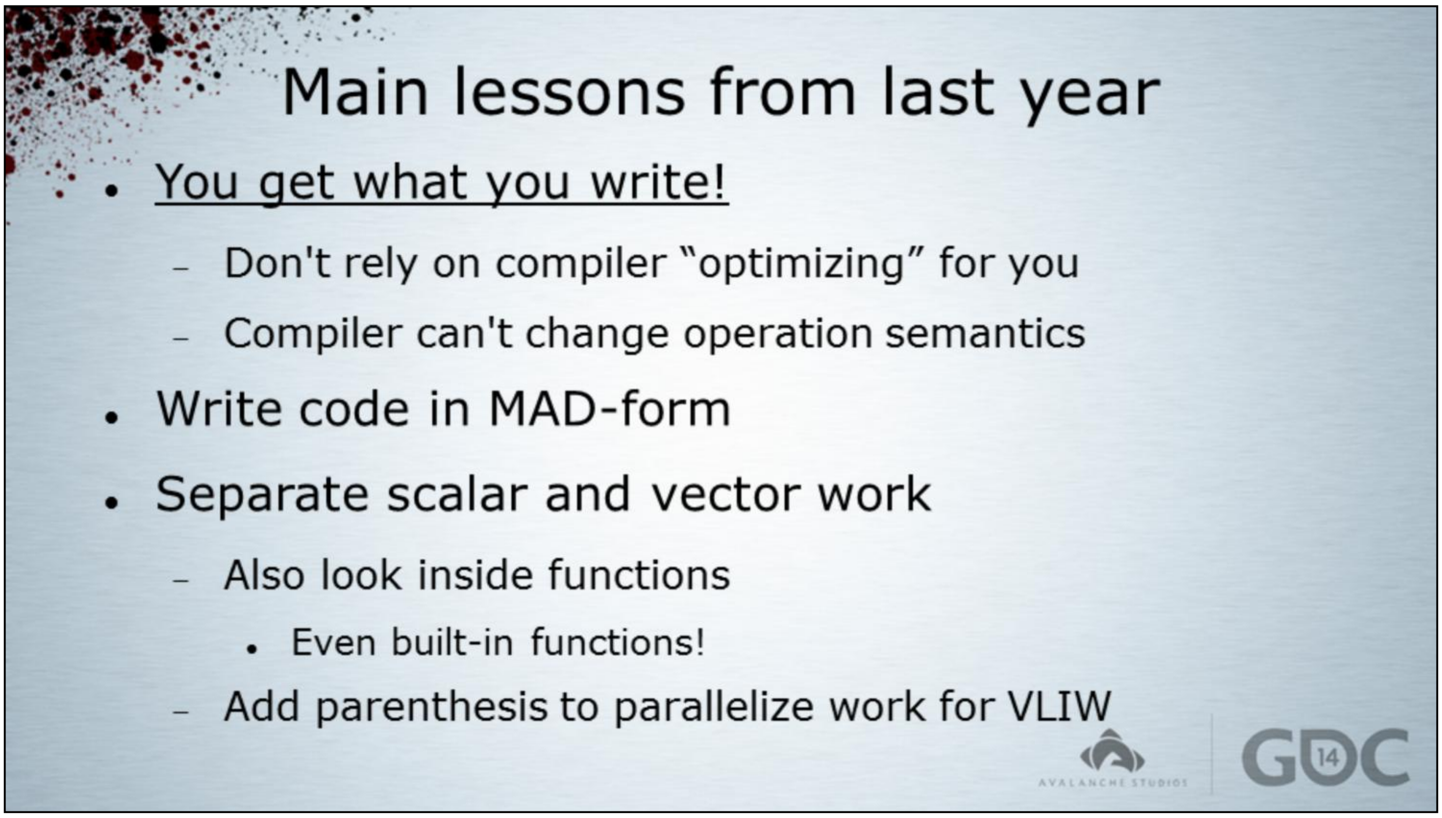
Low-level Shader Optimization for Next-Gen and DX11

Emil Persson
Head of Research, Avalanche Studios



Introduction

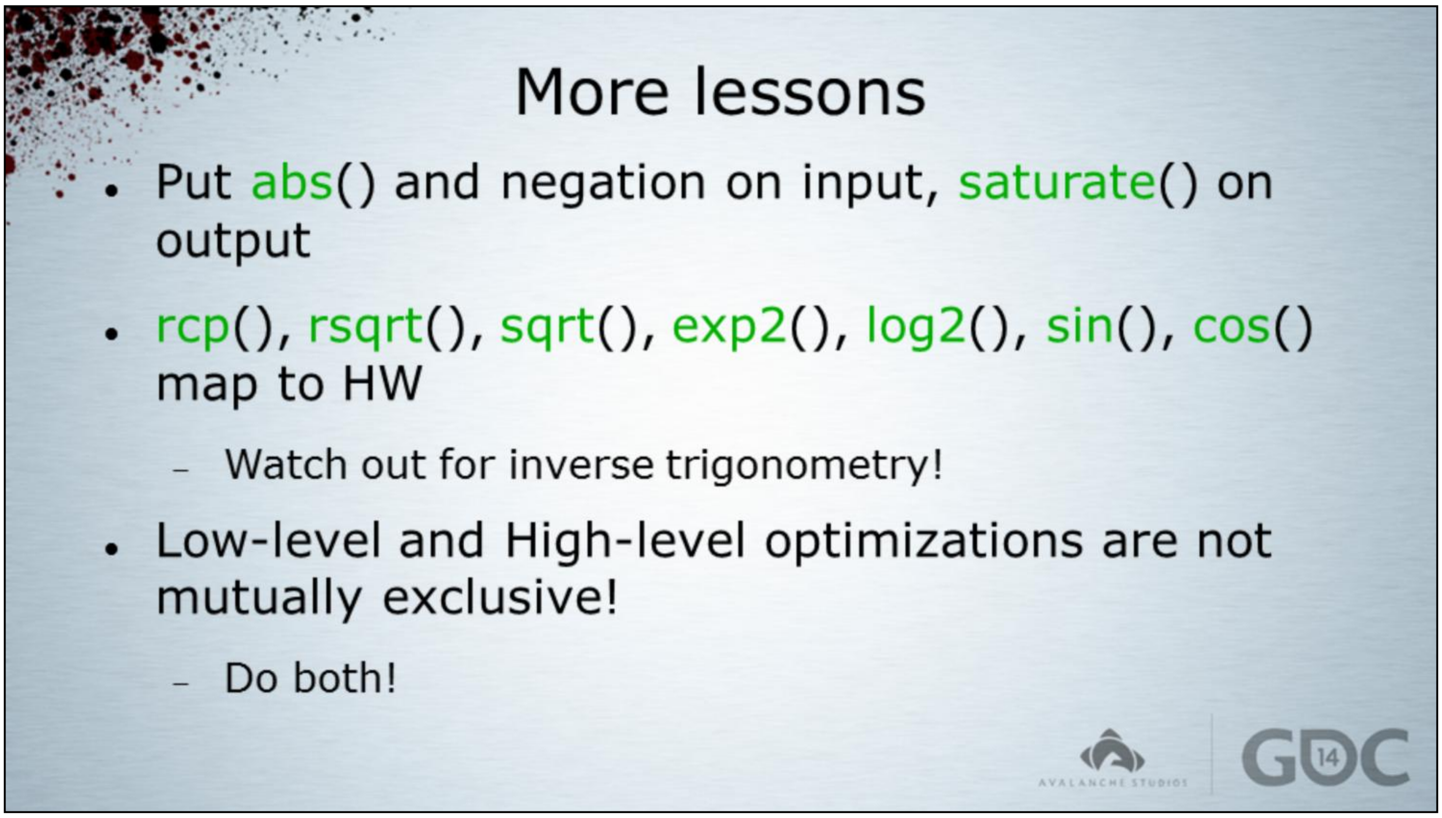
- Last year's talk
  - *"Low-level Thinking in High-level Shading Languages"*
  - Covered the basic shader features set
    - Float ALU ops
- New since last year
  - Next-gen consoles
    - GCN-based GPUs
  - DX11 feature set mainstream
    - 70% on Steam have DX11 GPUs [1]



Main lessons from last year

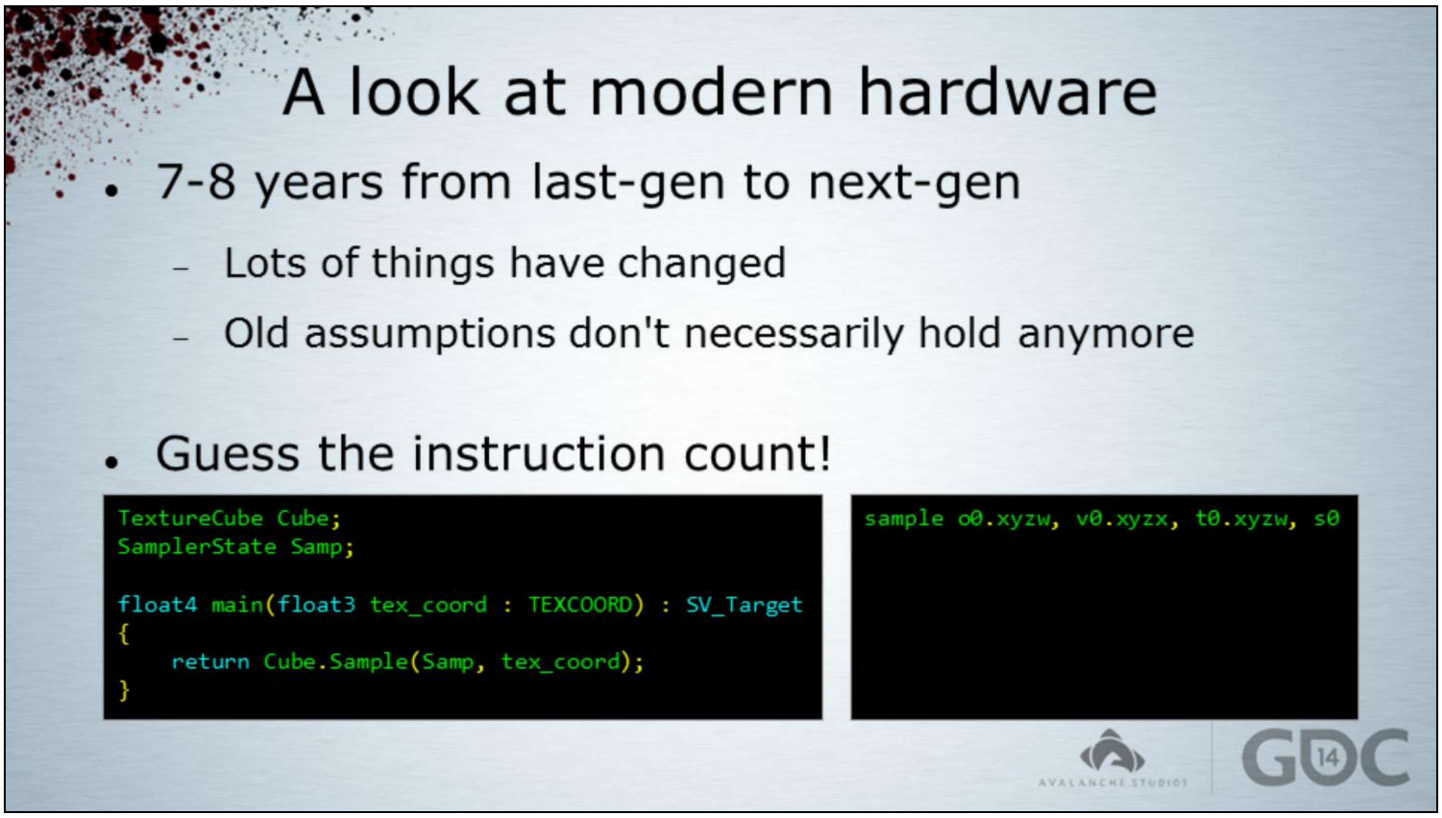
- You get what you write!
  - Don't rely on compiler “optimizing” for you
  - Compiler can't change operation semantics
- Write code in MAD-form
- Separate scalar and vector work
  - Also look inside functions
    - Even built-in functions!
  - Add parenthesis to parallelize work for VLIW

Refer to slide deck from last year (”Low-level Thinking in High-level Shading Languages”) for details on these optimizations. This presentation will assume that you already know what things like ”MAD-form” means.



More lessons

- Put abs() and negation on input, saturate() on output
- rcp(), rsqrt(), sqrt(), exp2(), log2(), sin(), cos() map to HW
  - Watch out for inverse trigonometry!
- Low-level and High-level optimizations are not mutually exclusive!
  - Do both!



A look at modern hardware

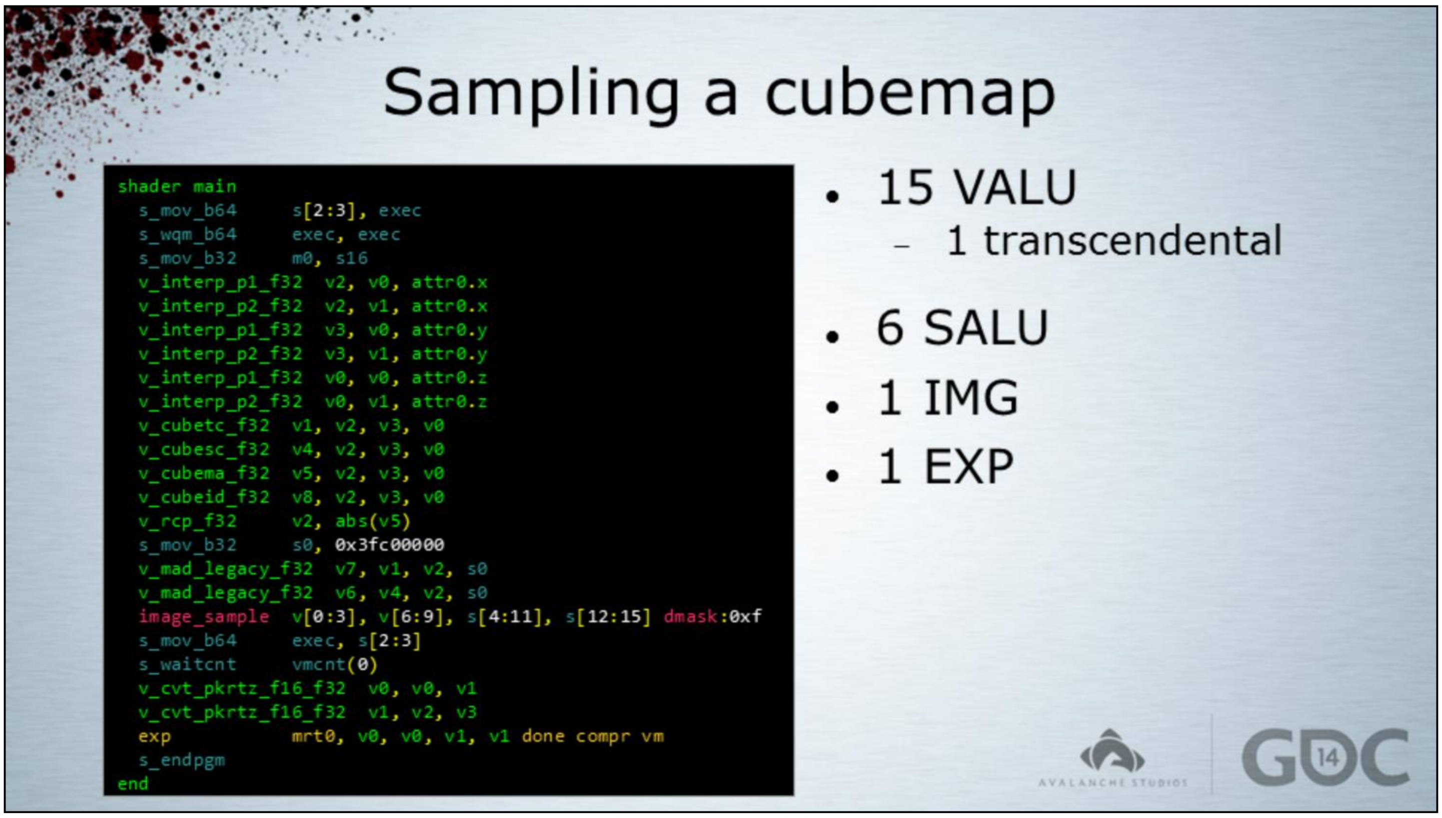
- 7-8 years from last-gen to next-gen
  - Lots of things have changed
  - Old assumptions don't necessarily hold anymore
- Guess the instruction count!

```
TextureCube Cube;
SamplerState Samp;

float4 main(float3 tex_coord : TEXCOORD) : SV_Target
{
    return Cube.Sample(Samp, tex_coord);
}
```

```
sample o0.xyzw, v0.xyzx, t0.xyzw, s0
```

Back in DX9 era a cubemap lookup was still a single sample instruction, and the same was true for projective textures (tex2Dproj). In DX10 direct support for projective textures was removed, with the expectation that shaders that need projective texturing will simply do the division by w manually. This reflected the fact that no hardware did the division by w in the texture unit anymore, so there was no need to pretend it did. The cost of this fixed function hardware could no longer be motivated when we have so much ALU units that would be perfectly capable of doing this math. The situation for cubemaps is the same. Obviously we still need fast sampling, so cubemaps are still a first class citizen in the API and will likely remain that way; however, the coordinate normalization is not something that we want to spend an awful lot of transistors on when those transistors could rather be used to add more general ALU cores instead. Consequently, this is handled by the ALUs these days. The D3D bytecode still treats sampling a cubemap as a simple sample instruction. However, it may surprise you what this expands to in native hardware instructions.



Sampling a cubemap

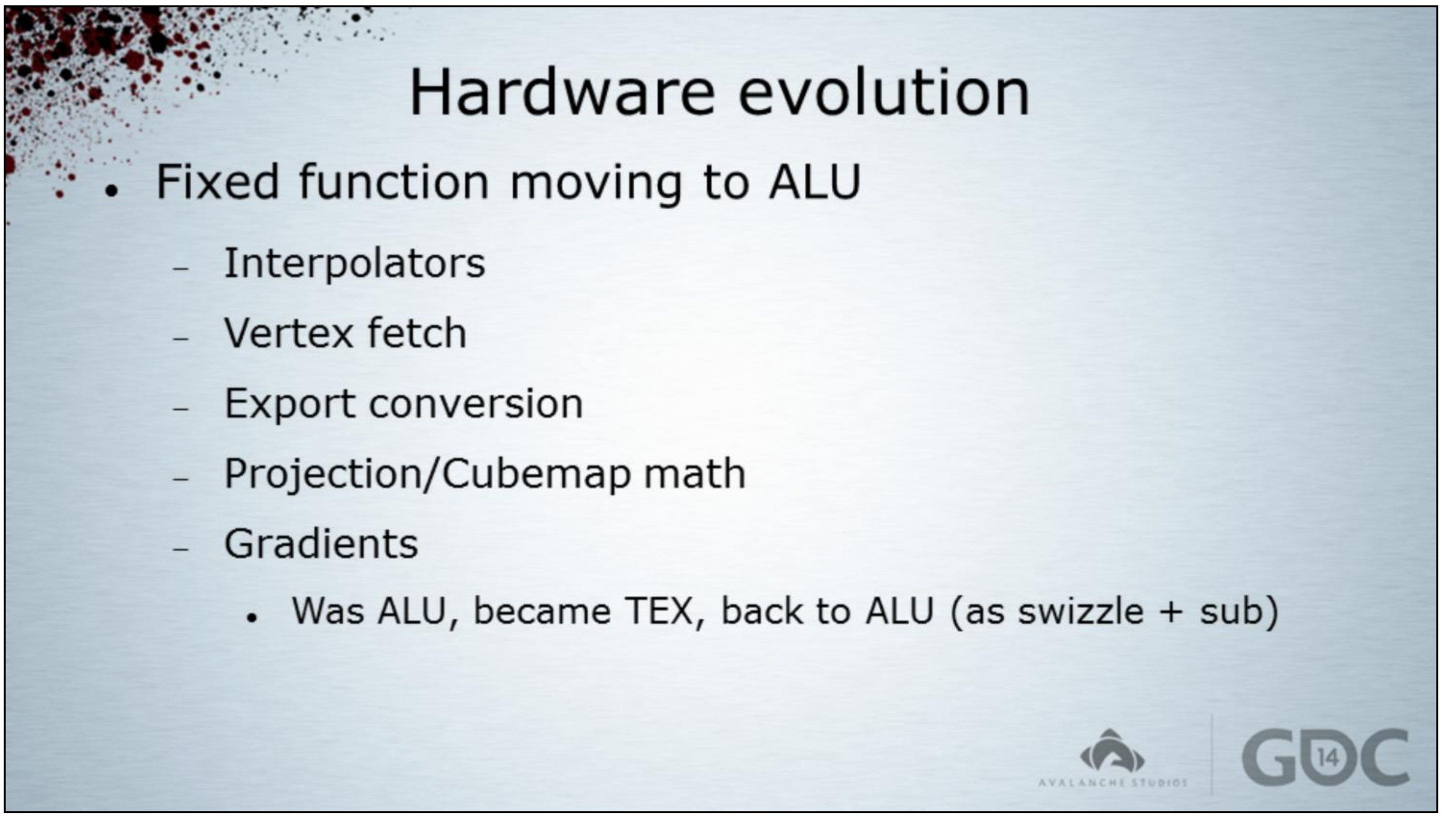
```
shader main
  s_mov_b64      s[2:3], exec
  s_wqm_b64      exec, exec
  s_mov_b32      m0, s16
  v_interp_p1_f32  v2, v0, attr0.x
  v_interp_p2_f32  v2, v1, attr0.x
  v_interp_p1_f32  v3, v0, attr0.y
  v_interp_p2_f32  v3, v1, attr0.y
  v_interp_p1_f32  v0, v0, attr0.z
  v_interp_p2_f32  v0, v1, attr0.z
  v_cubetc_f32   v1, v2, v3, v0
  v_cubesc_f32   v4, v2, v3, v0
  v_cubema_f32   v5, v2, v3, v0
  v_cubeid_f32   v8, v2, v3, v0
  v_rcp_f32      v2, abs(v5)
  s_mov_b32      s0, 0x3fc00000
  v_mad_legacy_f32  v7, v1, v2, s0
  v_mad_legacy_f32  v6, v4, v2, s0
  image_sample   v[0:3], v[6:9], s[4:11], s[12:15] dmask:0xf
  s_mov_b64      exec, s[2:3]
  s_waitcnt      vmcnt(0)
  v_cvt_pkrtz_f16_f32  v0, v0, v1
  v_cvt_pkrtz_f16_f32  v1, v2, v3
  exp            mrt0, v0, v0, v1, v1 done compr vm
  s_endpgm
end
```

- 15 VALU
  - 1 transcendental
- 6 SALU
- 1 IMG
- 1 EXP

AVALANCHE STUDIOS

GDC 14

This is what the actual shader looks like in the end. There is a set of different types of instructions here, vector ALU instruction (VALU) which are your typical math instructions and operate on wide SIMD vectors across all threads/pixels/vertices, and scalar instructions (SALU) that operate on things that are common for all threads. More on these instructions later in this presentation. There is also an image instruction (IMG) that does the actual sampling here, and finally an export instruction (EXP) that writes out the final output data, which in the case of a pixel shader is what lands in your framebuffer.

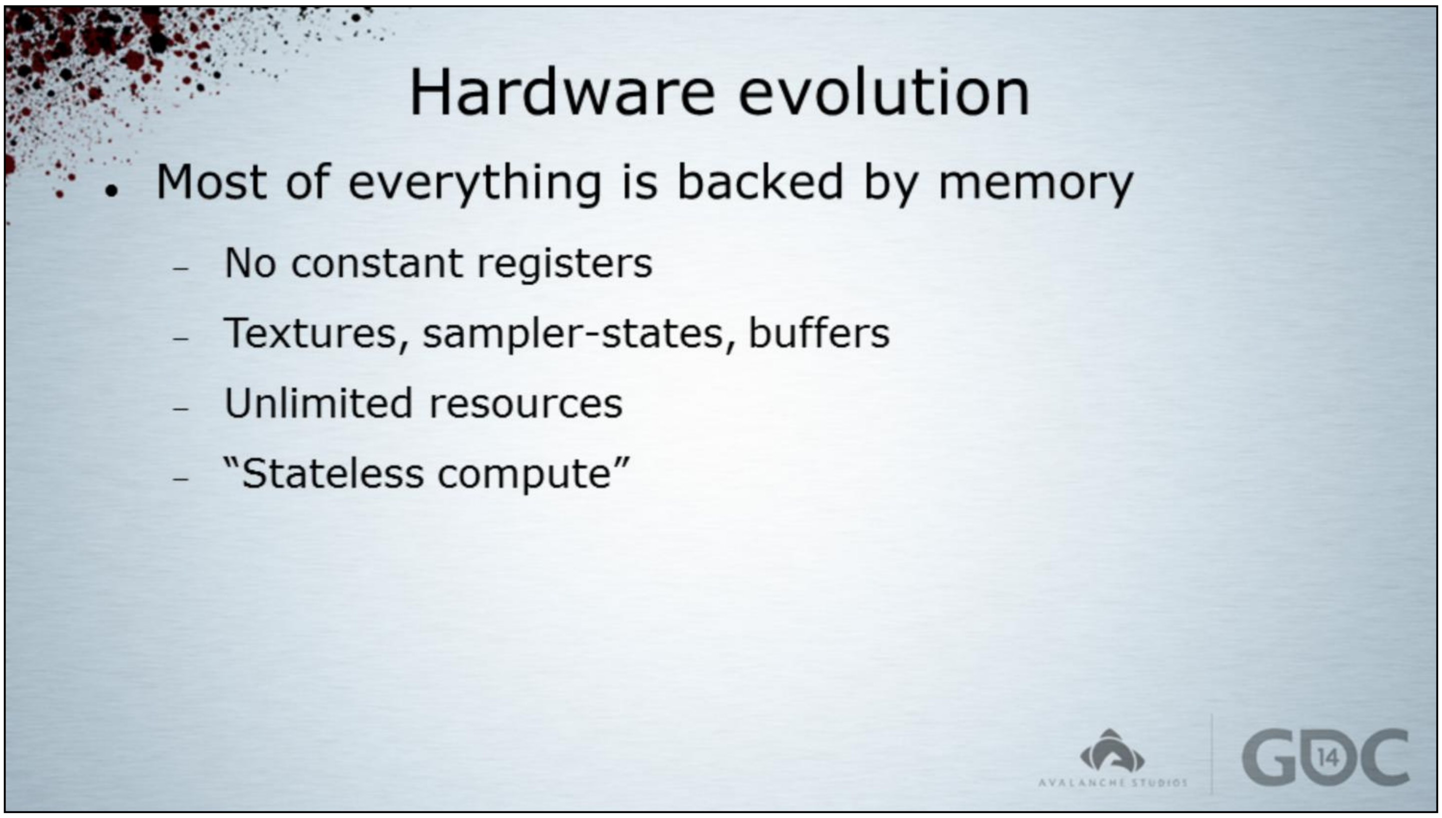


Hardware evolution

- Fixed function moving to ALU
  - Interpolators
  - Vertex fetch
  - Export conversion
  - Projection/Cubemap math
  - Gradients
    - Was ALU, became TEX, back to ALU (as swizzle + sub)

The general trend is that more and more fixed function units move over to the shader cores. This makes a lot of sense from a transistor budget point of view and is something that has been going on for a long time. Interpolators became ALU instructions with DX11 hardware. Vertex fetch has been done by the shader for a long time. Even Xbox360 did this. Export conversion is now handled by the ALUs since GCN. Projection/cubemap math since DX10. Gradients have moved a bit back and forth. Since gradient are needed by the texture units anyway, it made sense in the past to let them handle it; however, now that GCN has a generic lane swizzle, the ALUs has all the tools to do the work itself, so now it’s done in the ALUs again.

A side effect of this trend is that things that previously were more or less for free could now come at a moderate cost in terms of ALU instructions. For instance, for shaders with a sufficiently large number of instructions / interpolator ratio interpolators used to be free, although short shaders or shaders with many interpolators could easily become interpolator-bound. On DX11 hardware where interpolation is an ALU instruction, you basically pay for the interpolation cost. Previously interpolator-bound shaders could now became ALU-bound and likely run substantially faster, whereas if you weren’t previously interpolator-bound, you would now see a slowdown due to the additional interpolation cost.



Hardware evolution

- Most of everything is backed by memory
  - No constant registers
  - Textures, sampler-states, buffers
  - Unlimited resources
  - "Stateless compute"

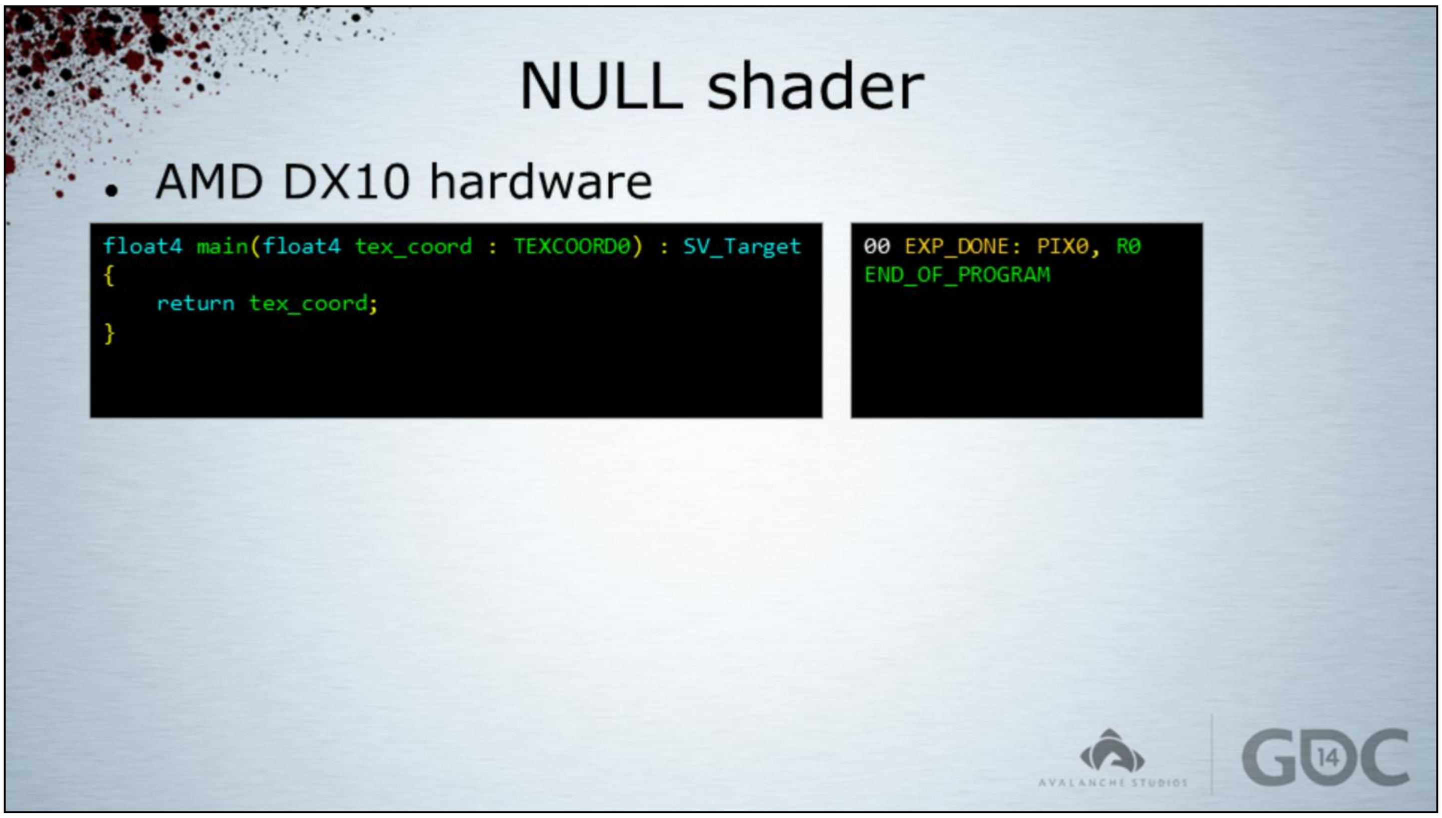
In the past the hardware had a lot of global device state. Things sat is registers that the hardware units read. This is not the case anymore. Most things are backed by memory and then read by the hardware whenever it needs it. This is not as scary as it may sound, there are obviously caches in-between keeping the bandwidth cost to a minimum, and values may be lying around in local registers for whatever hardware unit needs it after it has been loaded from memory. Constants obviously moved to memory with the introduction of constant buffers; however, these days things like texture descriptors and sampler-states are just a piece of data and behave just like constants. On GCN architecture you could technically put a bunch of texture descriptors and sampler-states in your constant buffer, provided of course that we have the proper API and shader infrastructure to do things that way.

For historical reasons APIs have assigned resources to “slots”. These do not exist in hardware anymore. Instead drivers assemble the set of active slots and create a list of those in memory and just passes the pointer to the shader.

The main implication of this is that there is no longer any particular limitation to the number of resources we can access from a shader. Just provide a long enough array and the shader can grab anything from anywhere as it pleases.

The other implication is that access to resources also comes at a cost for grabbing the backing data of sampler-states and texture descriptors. This cost can mostly be hidden on GCN since those are SALU instructions that run independently of VALU, but it is worth knowing that resource descriptors are loaded explicitly by the shader using actual shader instructions.

It is worth noting that compute units in GCN architecture are completely stateless. This means that once a compute shader is up and running, it has all data in its local registers and no longer depends on any global device state (other than data in memory). This means that it is perfectly possible for compute units to run in parallel with different shaders, completely asynchronously, and in parallel with the graphics pipeline. However, the graphics pipeline itself still relies on some global state, and thus it is currently not possible to run two different graphics pipelines in parallel. It would not surprise me if this would change in future hardware.



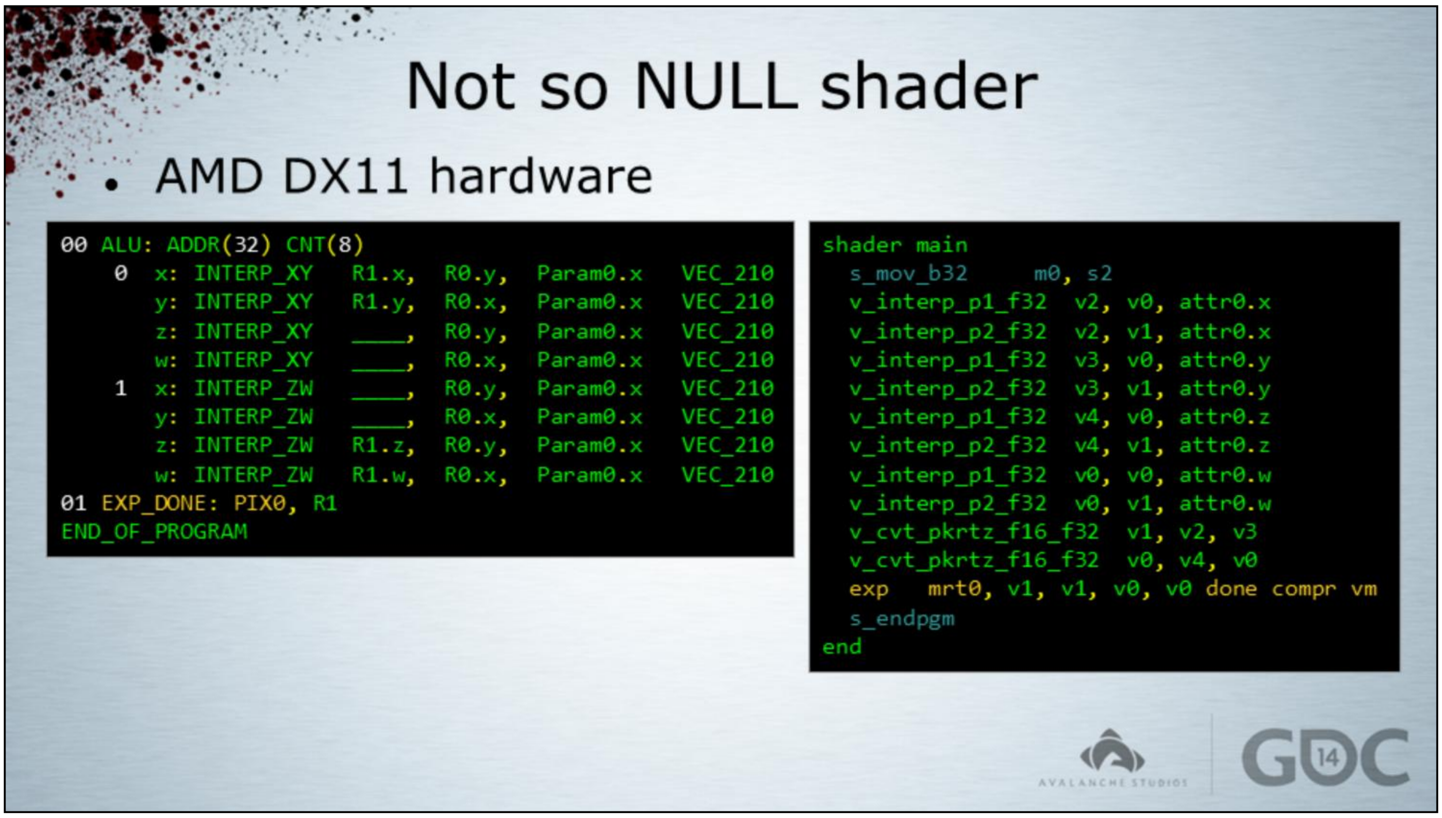
NULL shader

- AMD DX10 hardware

```
float4 main(float4 tex_coord : TEXCOORD0) : SV_Target
{
    return tex_coord;
}
```

```
00 EXP_DONE: PIX0, R0
END_OF_PROGRAM
```

An interesting exercise when trying to understand the basic parameters to a shader is trying to write a “NULL” shader, or a shader that outputs no actual instructions other than say a final export. On AMD’s DX10 level hardware something like this would do. We are simply returning an interpolator directly. On DX10 level hardware, interpolators came preloaded into registers, so we only need to output that register and we are done. One downside of this approach is that if you have a lot of interpolators the shader will by necessity also consume a lot of registers, just to provide the shader with its inputs. This can negatively impact latency hiding and ultimately performance, which is another reason to go away from this approach.



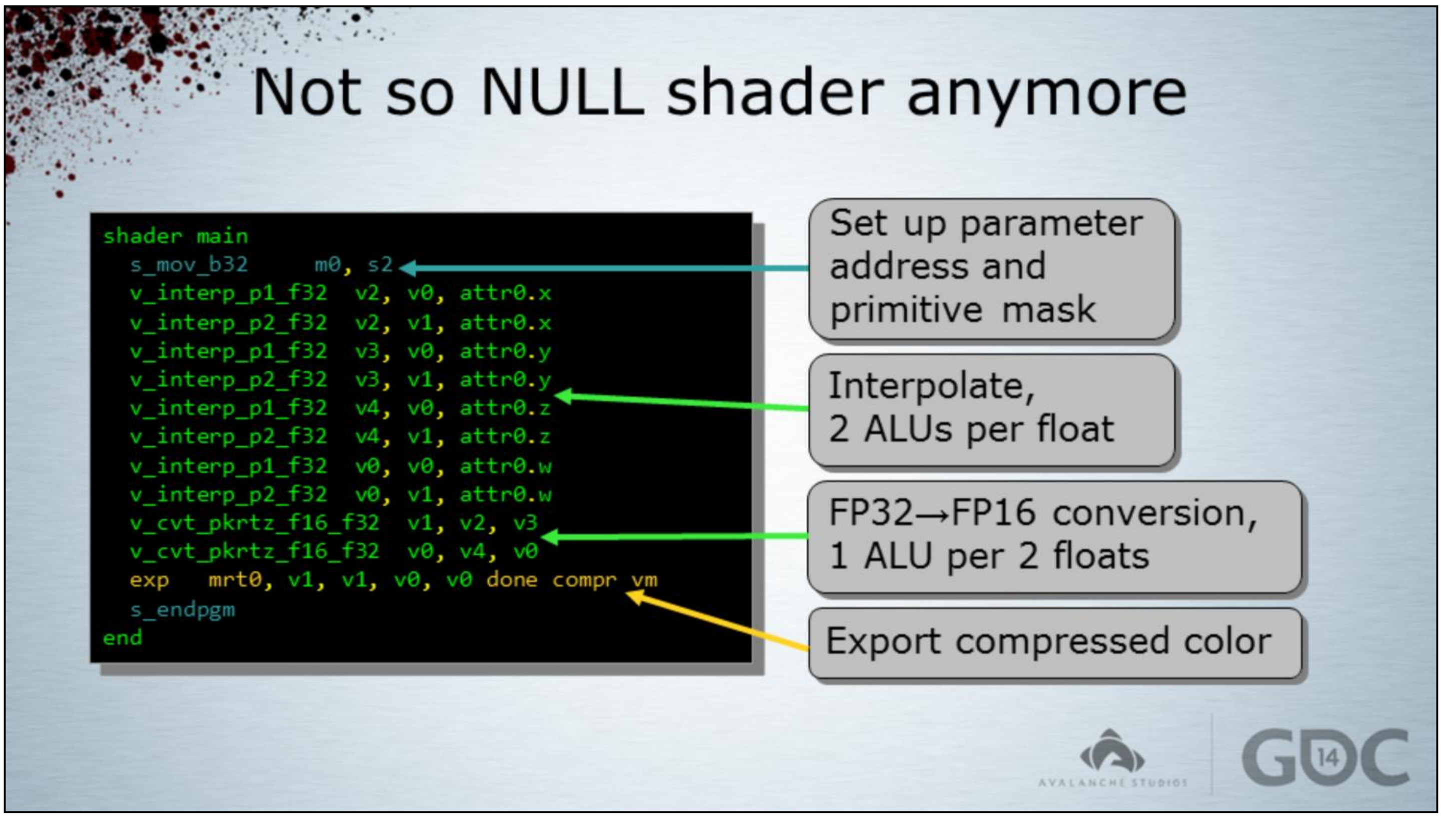
Not so NULL shader

- AMD DX11 hardware

```
00 ALU: ADDR(32) CNT(8)
    0  x: INTERP_XY   R1.x,   R0.y,   Param0.x    VEC_210
       y: INTERP_XY   R1.y,   R0.x,   Param0.x    VEC_210
       z: INTERP_XY   ____,   R0.y,   Param0.x    VEC_210
       w: INTERP_XY   ____,   R0.x,   Param0.x    VEC_210
    1  x: INTERP_ZW   ____,   R0.y,   Param0.x    VEC_210
       y: INTERP_ZW   ____,   R0.x,   Param0.x    VEC_210
       z: INTERP_ZW   R1.z,   R0.y,   Param0.x    VEC_210
       w: INTERP_ZW   R1.w,   R0.x,   Param0.x    VEC_210
01 EXP_DONE: PIX0, R1
END_OF_PROGRAM
```

```
shader main
  s_mov_b32     m0, s2
  v_interp_p1_f32  v2, v0, attr0.x
  v_interp_p2_f32  v2, v1, attr0.x
  v_interp_p1_f32  v3, v0, attr0.y
  v_interp_p2_f32  v3, v1, attr0.y
  v_interp_p1_f32  v4, v0, attr0.z
  v_interp_p2_f32  v4, v1, attr0.z
  v_interp_p1_f32  v0, v0, attr0.w
  v_interp_p2_f32  v0, v1, attr0.w
  v_cvt_pkrtz_f16_f32  v1, v2, v3
  v_cvt_pkrtz_f16_f32  v0, v4, v0
  exp   mrt0, v1, v1, v0, v0 done compr vm
  s_endpgm
end
```

On DX11 level hardware the situation is different. On all AMD DX11 hardware the interpolation is done manually by the shader. Left side is an HD5000 series chip, right side the results on the GCN architecture. On GCN we also see the addition of export conversion to pack the result into FP16 format.



Not so NULL shader anymore

```
shader main
  s_mov_b32      m0, s2
  v_interp_p1_f32  v2, v0, attr0.x
  v_interp_p2_f32  v2, v1, attr0.x
  v_interp_p1_f32  v3, v0, attr0.y
  v_interp_p2_f32  v3, v1, attr0.y
  v_interp_p1_f32  v4, v0, attr0.z
  v_interp_p2_f32  v4, v1, attr0.z
  v_interp_p1_f32  v0, v0, attr0.w
  v_interp_p2_f32  v0, v1, attr0.w
  v_cvt_pkrtz_f16_f32  v1, v2, v3
  v_cvt_pkrtz_f16_f32  v0, v4, v0
  exp   mrt0, v1, v1, v0, v0 done compr vm
  s_endpgm
end
```

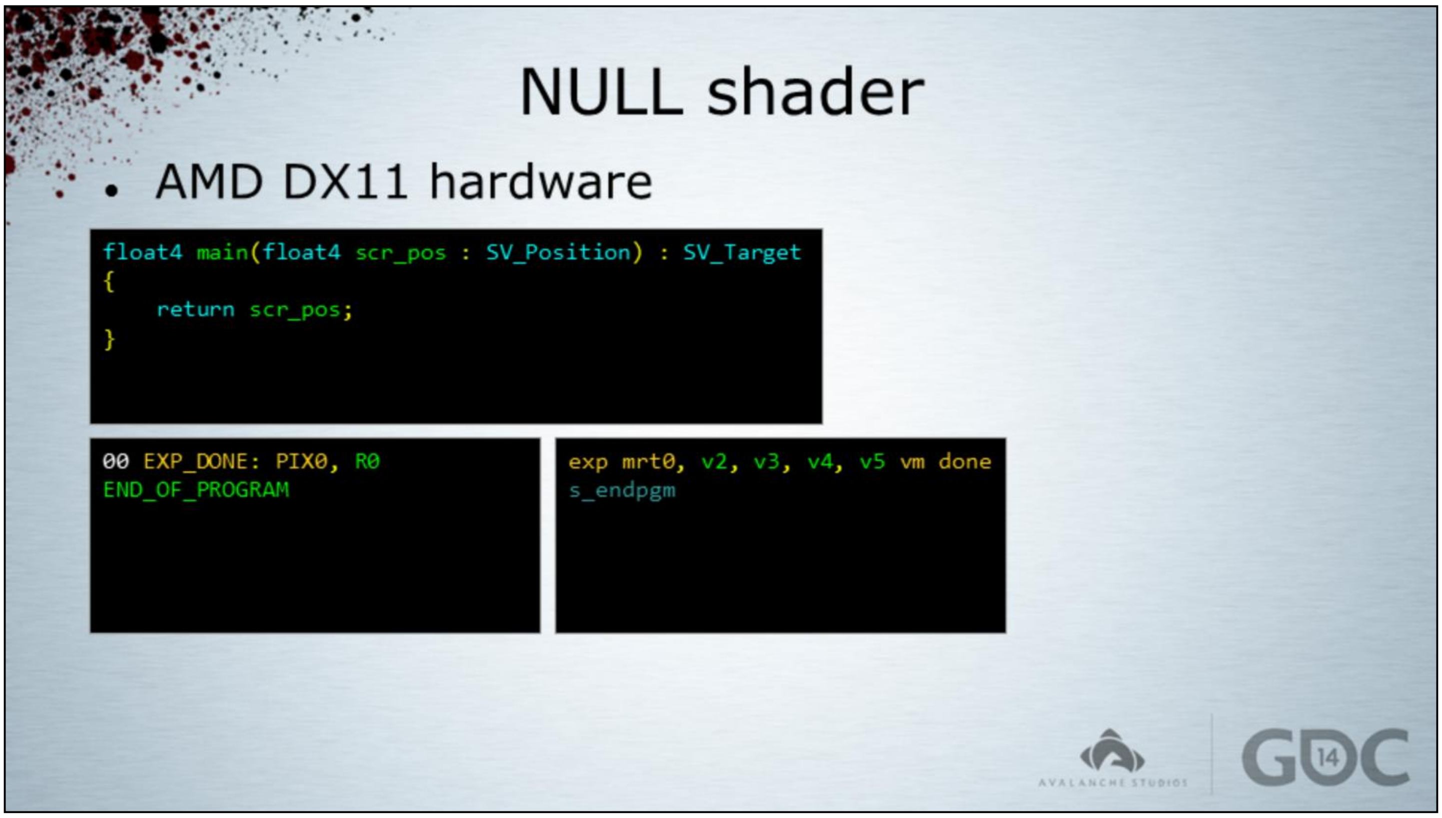
Set up parameter address and primitive mask

Interpolate, 2 ALUs per float

FP32→FP16 conversion, 1 ALU per 2 floats

Export compressed color

The simple pass-through shader from DX10 has now become a bunch of work to be done by the shader cores.



NULL shader

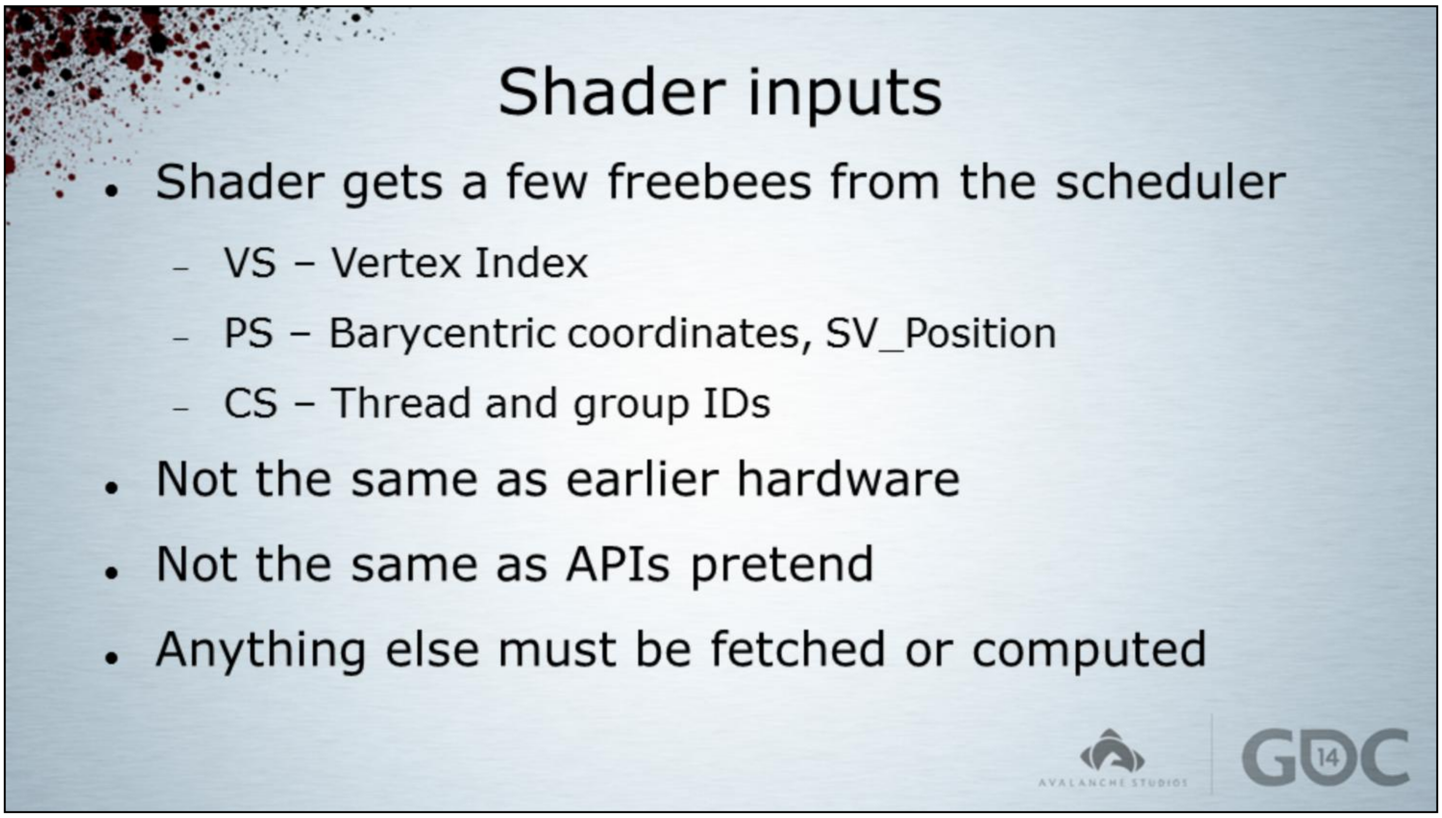
- AMD DX11 hardware

```
float4 main(float4 scr_pos : SV_Position) : SV_Target
{
    return scr_pos;
}
```

```
00 EXP_DONE: PIX0, R0
END_OF_PROGRAM
```

```
exp mrt0, v2, v3, v4, v5 vm done
s_endpgm
```

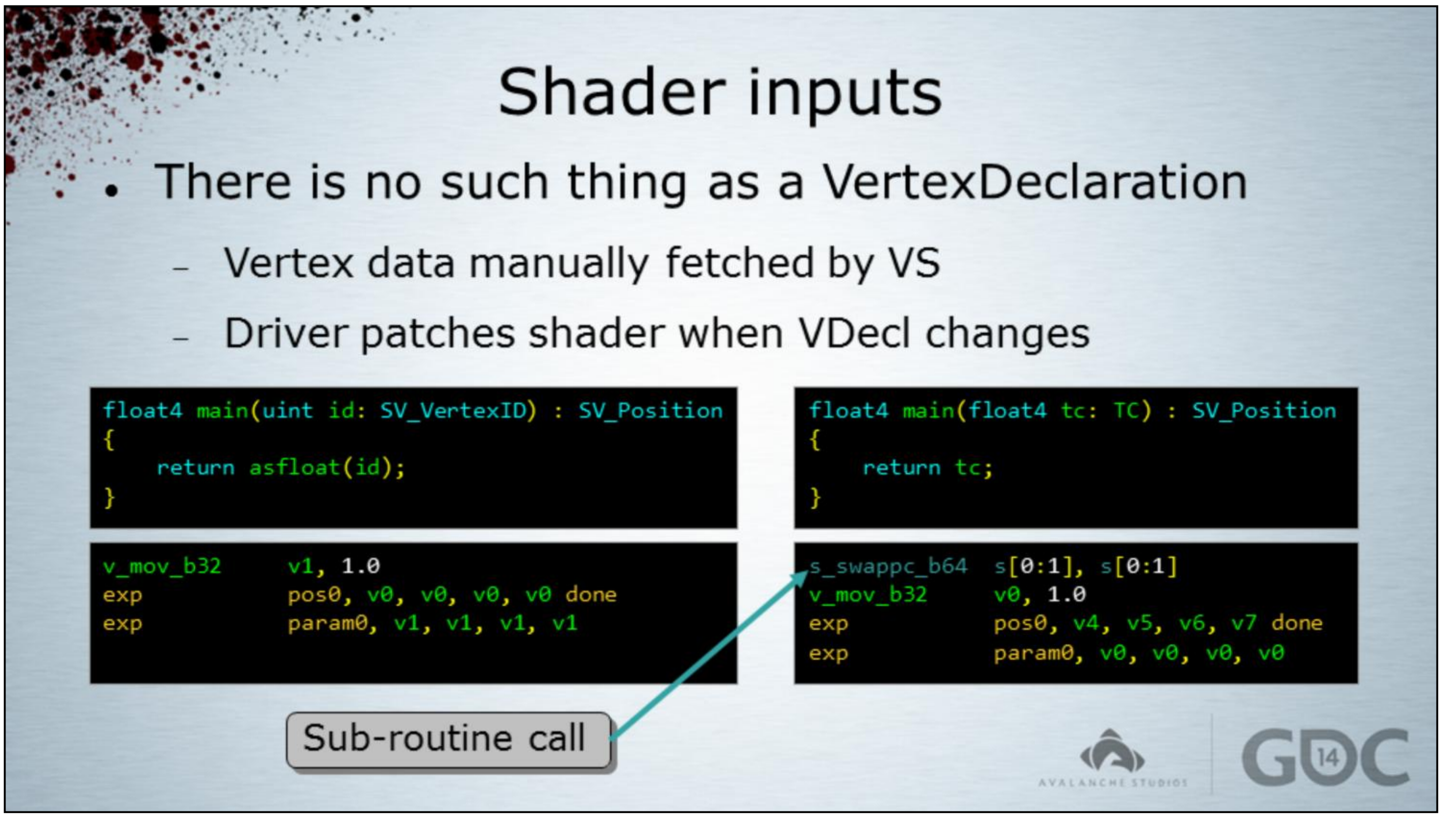
To create a NULL shader on DX11 one could simply return the screen position system value instead. These values still come preloaded in registers. On GCN we can see that they were loaded into registers v2-v5. The registers v0 and v1 usually hold the barycentric coordinates used for interpolation (not used here though).



Shader inputs

- Shader gets a few freebees from the scheduler
  - VS – Vertex Index
  - PS – Barycentric coordinates, SV_Position
  - CS – Thread and group IDs
- Not the same as earlier hardware
- Not the same as APIs pretend
- Anything else must be fetched or computed

A shader only gets a handful of input parameters to work with, the rest of the data it needs must be loaded or computed. Depending on what shader type we are talking about, the input parameter varies. A vertex shader, for instance, gets a vertex index. This is used to manually fetch the vertex data. A pixel shaders gets barycentric coordinates for interpolation, as well as the screen position. A compute shader gets thread and group IDs. From the way shaders look in HLSL it may appear as if some things are for free from the shader’s point of view, such as vertex data, however, there is an increasingly large gap between what the APIs pretend and what the underlying hardware does.



Shader inputs

- There is no such thing as a VertexDeclaration
  - Vertex data manually fetched by VS
  - Driver patches shader when VDecl changes

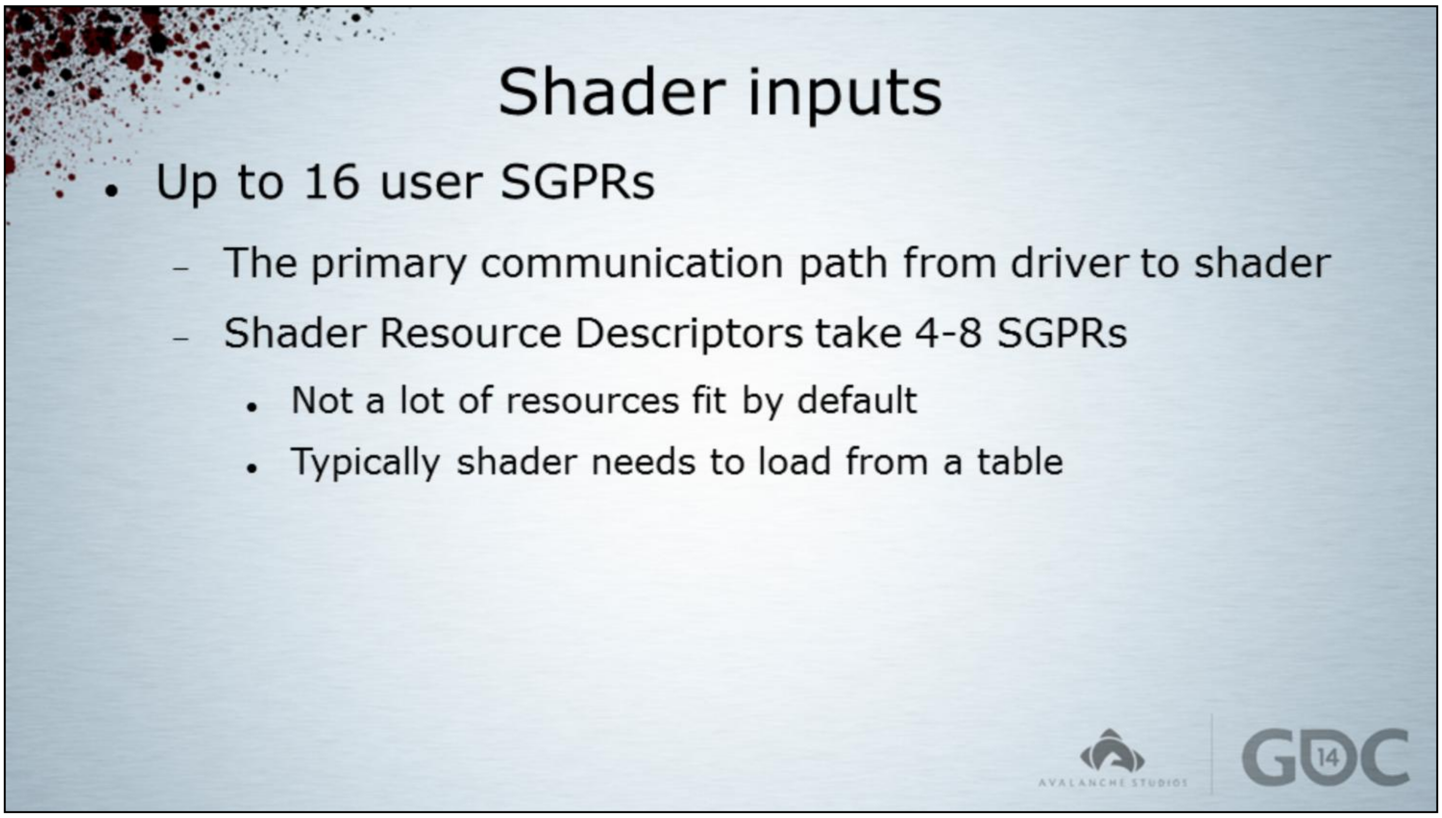
```
float4 main(uint id: SV_VertexID) : SV_Position
{
    return asfloat(id);
}
```

```
v_mov_b32      v1, 1.0
exp            pos0, v0, v0, v0, v0 done
exp            param0, v1, v1, v1, v1
```

```
float4 main(float4 tc: TC) : SV_Position
{
    return tc;
}
```

```
s_swappc_b64   s[0:1], s[0:1]
v_mov_b32      v0, 1.0
exp            pos0, v4, v5, v6, v7 done
exp            param0, v0, v0, v0, v0
```

Vertex fetch has not been a fixed function hardware unit for quite some time, despite what APIs pretend. Even good old Xbox360 did vertex fetch manually in the shader. The vertex declaration is an API construct that does not map to any underlying hardware object. Instead a the shader is patched according to the current vertex declaration. On Xbox360 this could be very expensive since the vertex fetch was inserted as a bunch of instruction directly into the top of the vertex shader instruction stream. For GCN architecture it can be done in a simpler way with a simple sub-routing call. Depending on what the active vertex declaration is, the driver provides a function pointer in two scalar registers that the shader simply calls as the first thing in the vertex shader. Note that in the left shader where we do not use any vertex streams, the function call is absent.

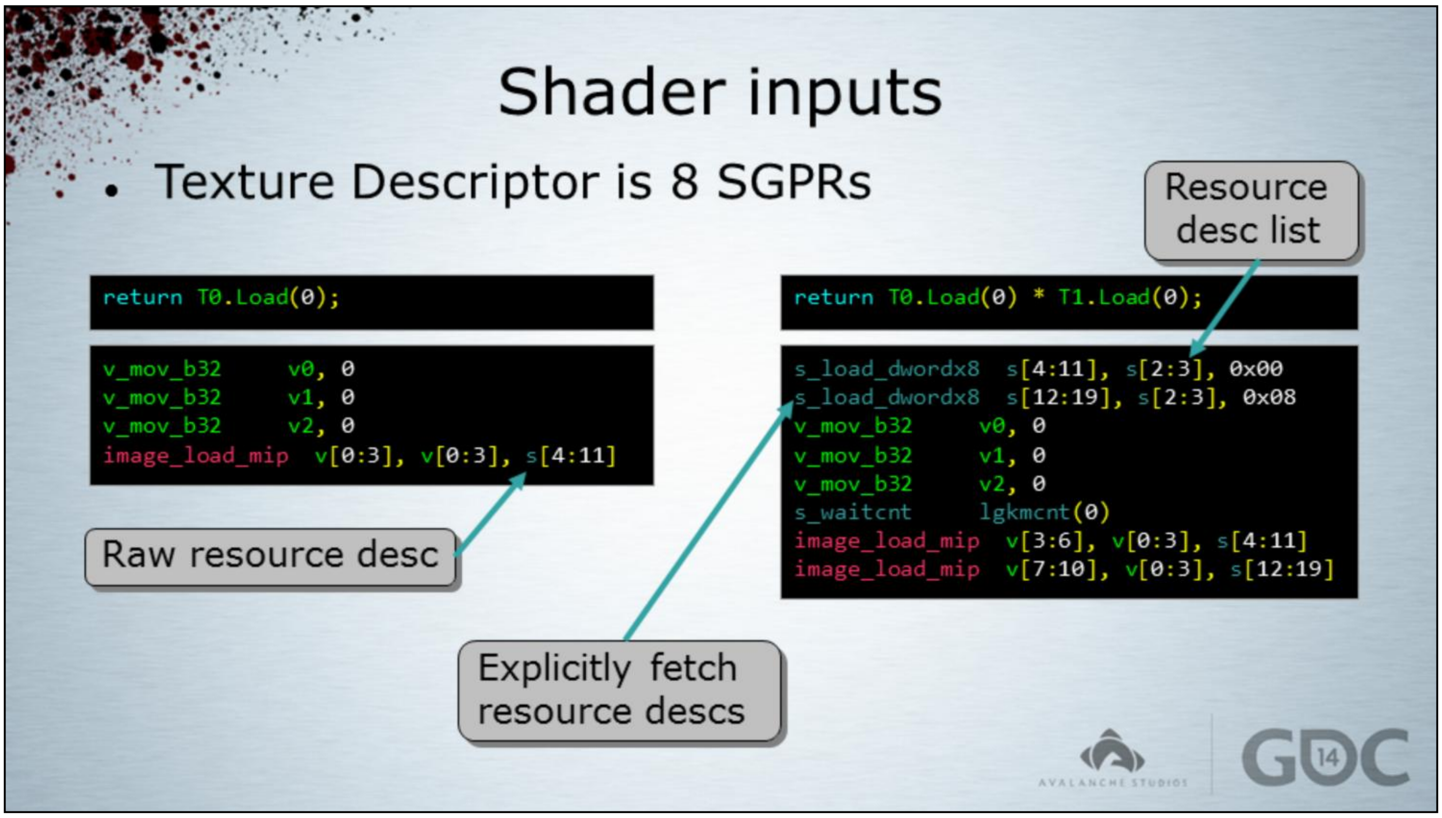


Shader inputs

- Up to 16 user SGPRs
  - The primary communication path from driver to shader
  - Shader Resource Descriptors take 4-8 SGPRs
    - Not a lot of resources fit by default
    - Typically shader needs to load from a table

AVALANCHE STUDIOS
GDC 14

Other than the hardware provided data, the driver can also pass a small set of data into the shader. Typically the driver will pass information such as a pointer to the list of resources. If the number of resources is small enough to fit within the maximum 16 scalar registers, the driver will simply pass the raw resource descriptor and avoid an indirection. Given the limited amount of data possible to pass, this only works in a very limited number of cases, such as using only a single texture.



Shader inputs

- Texture Descriptor is 8 SGPRs

```
return T0.Load(0);
```

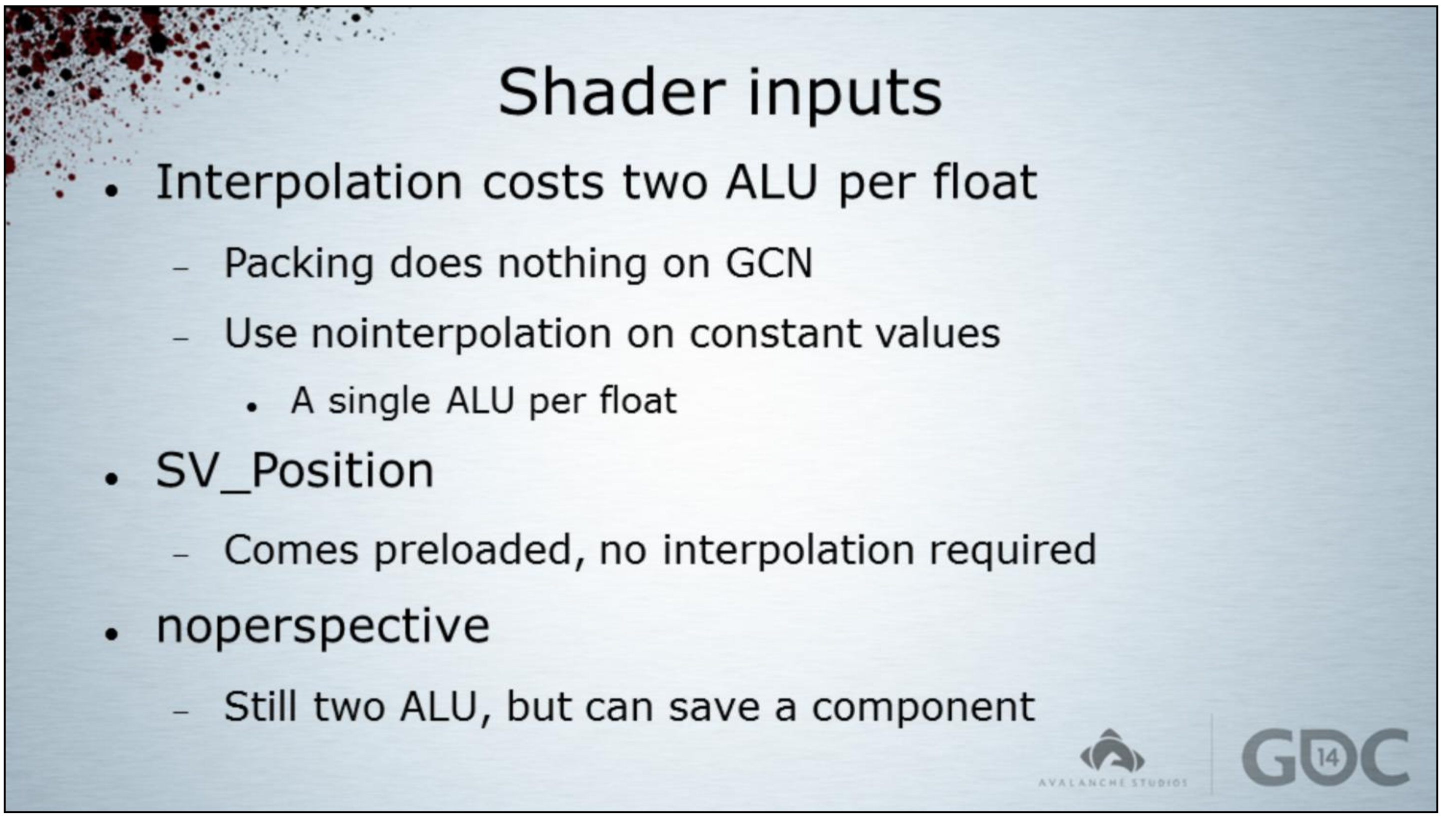
```
v_mov_b32      v0, 0
v_mov_b32      v1, 0
v_mov_b32      v2, 0
image_load_mip  v[0:3], v[0:3], s[4:11]
```

```
return T0.Load(0) * T1.Load(0);
```

```
s_load_dwordx8  s[4:11], s[2:3], 0x00
s_load_dwordx8  s[12:19], s[2:3], 0x08
v_mov_b32      v0, 0
v_mov_b32      v1, 0
v_mov_b32      v2, 0
s_waitcnt      lgkmcnt(0)
image_load_mip  v[3:6], v[0:3], s[4:11]
image_load_mip  v[7:10], v[0:3], s[12:19]
```

Raw resource desc
Resource desc list
Explicitly fetch resource descs
AVALANCHE STUDIOS
GDC 14

In the case of a single texture, the shader can use a raw resource descriptor. With two textures, the driver has to create a list of texture descriptors in memory and provide a pointer to it. Then the shader must load the texture descriptors from that table before it can access the textures.



Shader inputs

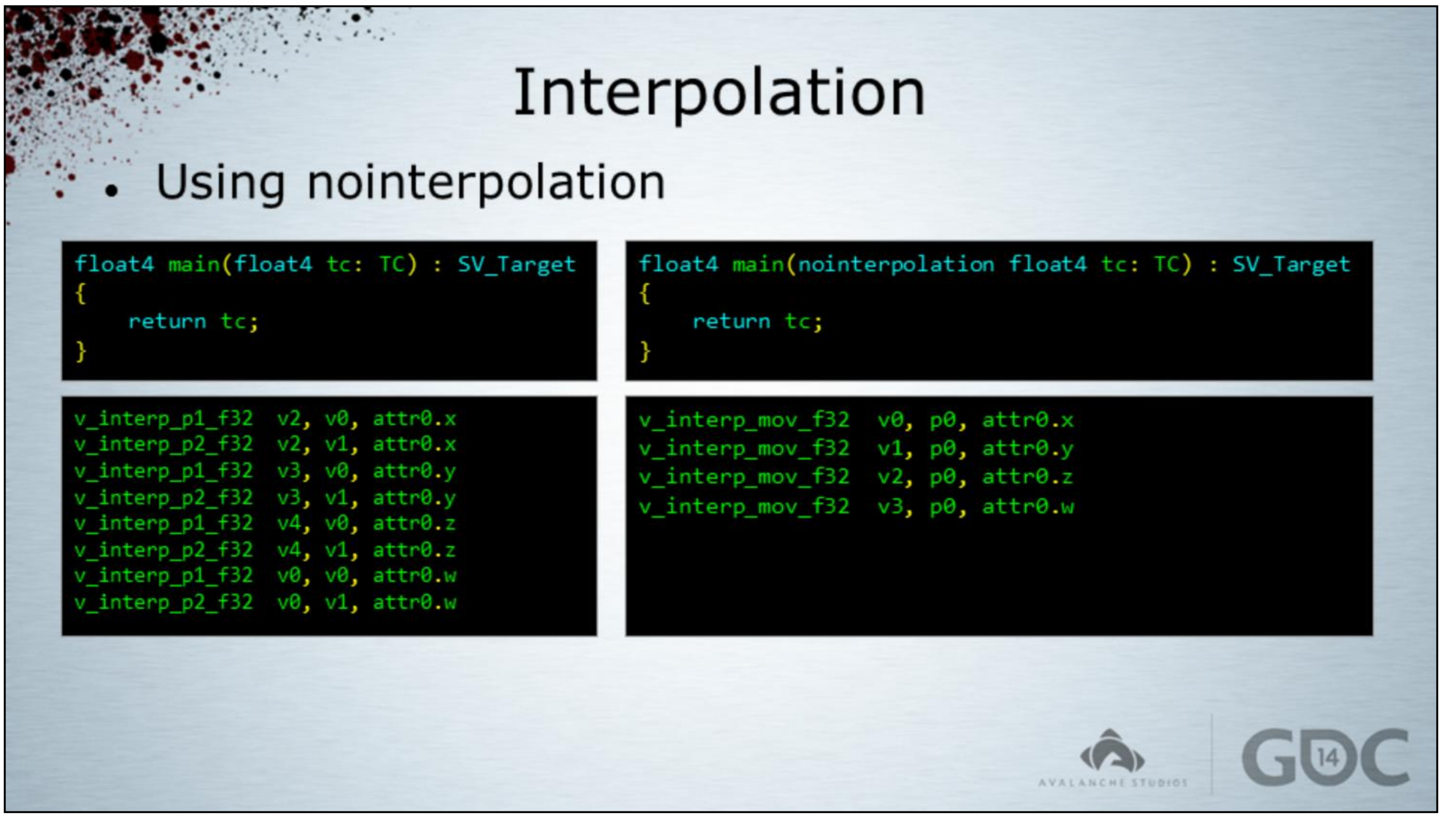
- Interpolation costs two ALU per float
  - Packing does nothing on GCN
  - Use nointerpolation on constant values
    - A single ALU per float
- SV_Position
  - Comes preloaded, no interpolation required
- noperspective
  - Still two ALU, but can save a component

AVALANCHE STUDIOS
GDC 14

In the past, an optimization that was commonly used was to pack interpolators together to reduce the number of interpolators, for instance packing two float2 texture coordinates into a single float4 vector. This used to be beneficial on older hardware, but on GCN this accomplishes nothing. It will still cost exactly the same, i.e. two ALU instructions per float. What will help though is to use the nointerpolation flag on attributes that are constant. This will reduce the cost of bringing the data in to a single ALU instruction since the data can just be copied into the shader instead of interpolated.

The position system value comes preloaded into shader registers, so from the shader’s point of view they are free. However, much like how you in the past could be interpolator-bound, you can also be bound by the generation of this system value by the hardware. So for short shaders using SV_Position could slow things down, whereas for longer shaders they become free and could avoid the cost of interpolation for screen-coordinate values.

The noperspective attribute doesn’t really affect the interpolation cost. It just means different barycentric coordinates are provided. It still costs 2 ALUs per float. However, in some cases where in the past you may have passed a w value and done the division by that in the pixel shader, you can instead use the noperspective flags and save one component as well as skip the division, so it could still save up to 9 ALU instructions for an xyz(w) attribute.



Interpolation

- Using nointerpolation

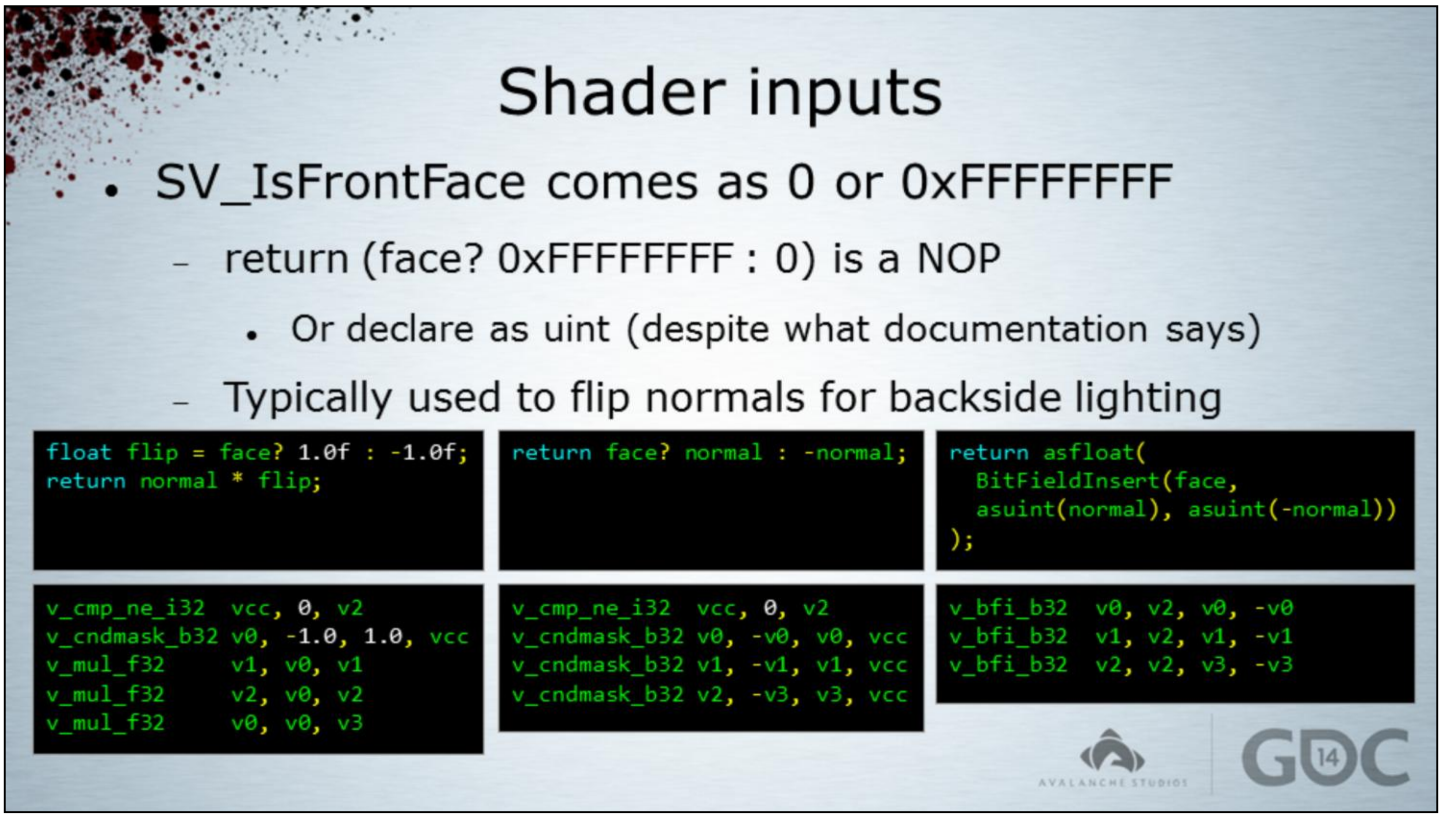
```
float4 main(float4 tc: TC) : SV_Target
{
    return tc;
}
```

```
float4 main(nointerpolation float4 tc: TC) : SV_Target
{
    return tc;
}
```

```
v_interp_p1_f32  v2, v0, attr0.x
v_interp_p2_f32  v2, v1, attr0.x
v_interp_p1_f32  v3, v0, attr0.y
v_interp_p2_f32  v3, v1, attr0.y
v_interp_p1_f32  v4, v0, attr0.z
v_interp_p2_f32  v4, v1, attr0.z
v_interp_p1_f32  v0, v0, attr0.w
v_interp_p2_f32  v0, v1, attr0.w
```

```
v_interp_mov_f32  v0, p0, attr0.x
v_interp_mov_f32  v1, p0, attr0.y
v_interp_mov_f32  v2, p0, attr0.z
v_interp_mov_f32  v3, p0, attr0.w
```

This is the effect of using nointerpolation, cutting the instruction count into half over interpolated values.



Shader inputs

- SV_IsFrontFace comes as 0 or 0xFFFFFFFF
  - return (face? 0xFFFFFFFF : 0) is a NOP
    - Or declare as uint (despite what documentation says)
  - Typically used to flip normals for backside lighting

```
float flip = face? 1.0f : -1.0f;
return normal * flip;
```

```
return face? normal : -normal;
```

```
return asfloat(
  BitFieldInsert(face,
  asuint(normal), asuint(-normal))
);
```

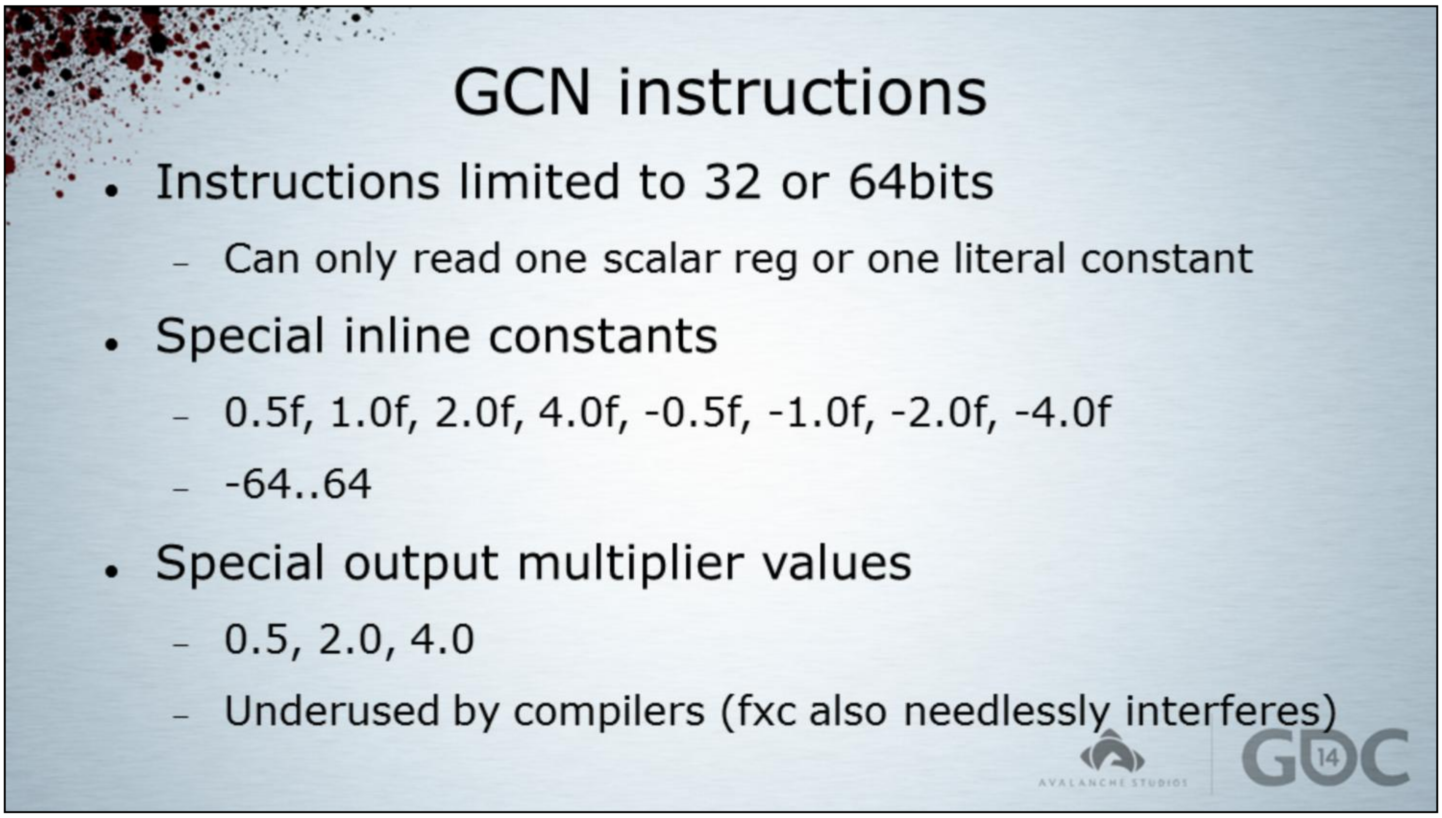
```
v_cmp_ne_i32  vcc, 0, v2
v_cndmask_b32 v0, -1.0, 1.0, vcc
v_mul_f32     v1, v0, v1
v_mul_f32     v2, v0, v2
v_mul_f32     v0, v0, v3
```

```
v_cmp_ne_i32  vcc, 0, v2
v_cndmask_b32 v0, -v0, v0, vcc
v_cndmask_b32 v1, -v1, v1, vcc
v_cndmask_b32 v2, -v3, v3, vcc
```

```
v_bfi_b32  v0, v2, v0, -v0
v_bfi_b32  v1, v2, v1, -v1
v_bfi_b32  v2, v2, v3, -v3
```

All API documentation consider SV_IsFrontFace to be a bool attribute. The hardware does not really have bools, all registers are 32bits and that is the smallest basic unit a GPU works with. So obviously bool attributes come at a full 32bit under the hood. Despite documentation, you can actually declare the SV_IsFrontFace variable as an uint and get the raw bits from the hardware, which is either 0x00000000 or 0xFFFFFFFF. This in combination with GCN's BFI instruction can be used to implement a very fast normal flipping for back-face lighting, which is the typical use case of this attribute.

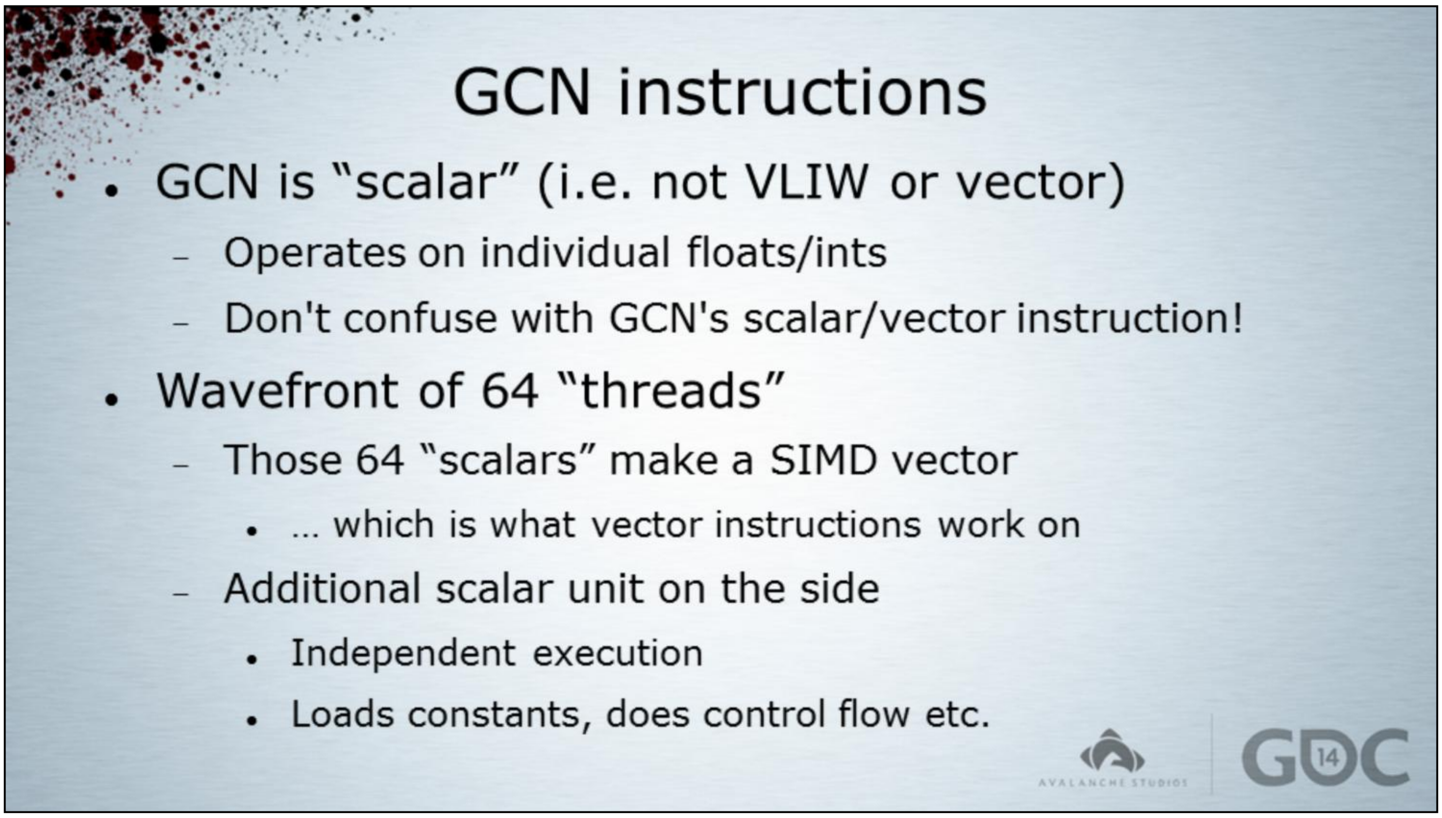
Left side shows a typical DX9/lastgen-esque implementation, which results in 5 instructions required. The middle one is a more DX10-ish implementation, which reduces the instruction count by one. Finally the right side shows the GCN-specific implementation. The BFI instruction basically does a bitwise blend between two inputs based on the mask. Given that the input SV_IsFrontFace flag is a bitmask of all zeros or ones, it becomes a select between the inputs. This allows us to drop the comparison, further reducing the instruction count by one.



GCN instructions

- Instructions limited to 32 or 64bits
  - Can only read one scalar reg or one literal constant
- Special inline constants
  - 0.5f, 1.0f, 2.0f, 4.0f, -0.5f, -1.0f, -2.0f, -4.0f
  - -64..64
- Special output multiplier values
  - 0.5, 2.0, 4.0
  - Underused by compilers (fxc also needlessly interferes)

The GCN architecture is a fair bit more restricted than earlier AMD hardware. The stated goal has been to reduce complexity to allow for more efficient hardware. The downside is that this in some cases leads to longer shaders than what you would see on earlier hardware. Given instructions size fixed at either 32 or 64bit, there is obviously not much space left for passing literal constants. Passing only one constant takes 32bits, so obviously there is no possibility at all to pass two given that we need some bits to encode the actual instruction as well. Additionally, the hardware can also only read a single scalar register per instruction, and not at the same time as a literal constant. This can be problematic for taking full advantage of the MAD instruction, which will be discussed later. There is however a limited set of common values, a handful powers of two float values and integers from -64 to 64 that have special encoding and can be provided for free. These can mitigate the problem in some cases. Similarly there are a few special output multipliers that may also help. Unfortunately compilers do not take advantage of this as much as they could, but that is at least a software problem and fixable.

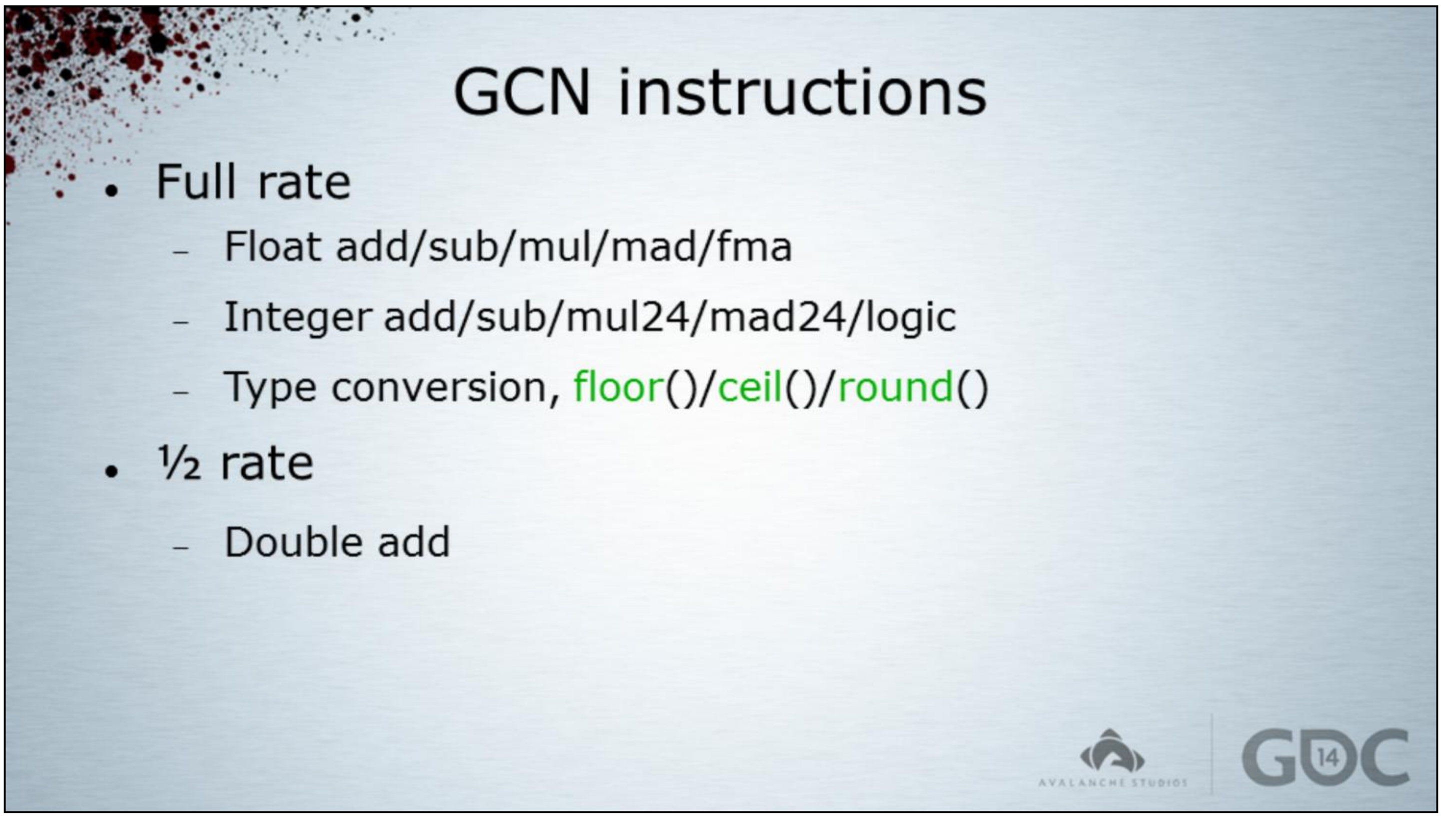


GCN instructions

- GCN is “scalar” (i.e. not VLIW or vector)
  - Operates on individual floats/ints
  - Don't confuse with GCN's scalar/vector instruction!
- Wavefront of 64 “threads”
  - Those 64 “scalars” make a SIMD vector
    - … which is what vector instructions work on
  - Additional scalar unit on the side
    - Independent execution
    - Loads constants, does control flow etc.

AVALANCHE STUDIOS
GDC 14

The GCN architecture is ”scalar”. This is in contrast with earlier VLIW and vector based architectures. This terminology can at first be a bit confusing when you hear about vector and scalar instructions, but these terms are viewing the hardware from different angles. From the shader’s point of view each instruction operates on a single float or integer. That is what “scalar” means when discussing the architecture. However, the hardware will still run many instances of the shader in lockstep, basically as a very wide SIMD vector, 64 wide to be precise in the case of GCN, and that is what we refer to as vector instructions. So where the shader programmer sees a scalar float, the hardware sees a 64 float wide SIMD vector. On the side of the wide vector ALU hangs a small scalar unit. This unit runs what is called scalar instructions. It does things that are common for all threads/shaders that runs in parallel, such as loading constants, fetching resource descriptors, as well as flow control. This unit runs independently of the shader vector math. So in typical shaders where the math or texture fetch is dominating the shading work, the scalar unit more or less operates for free.



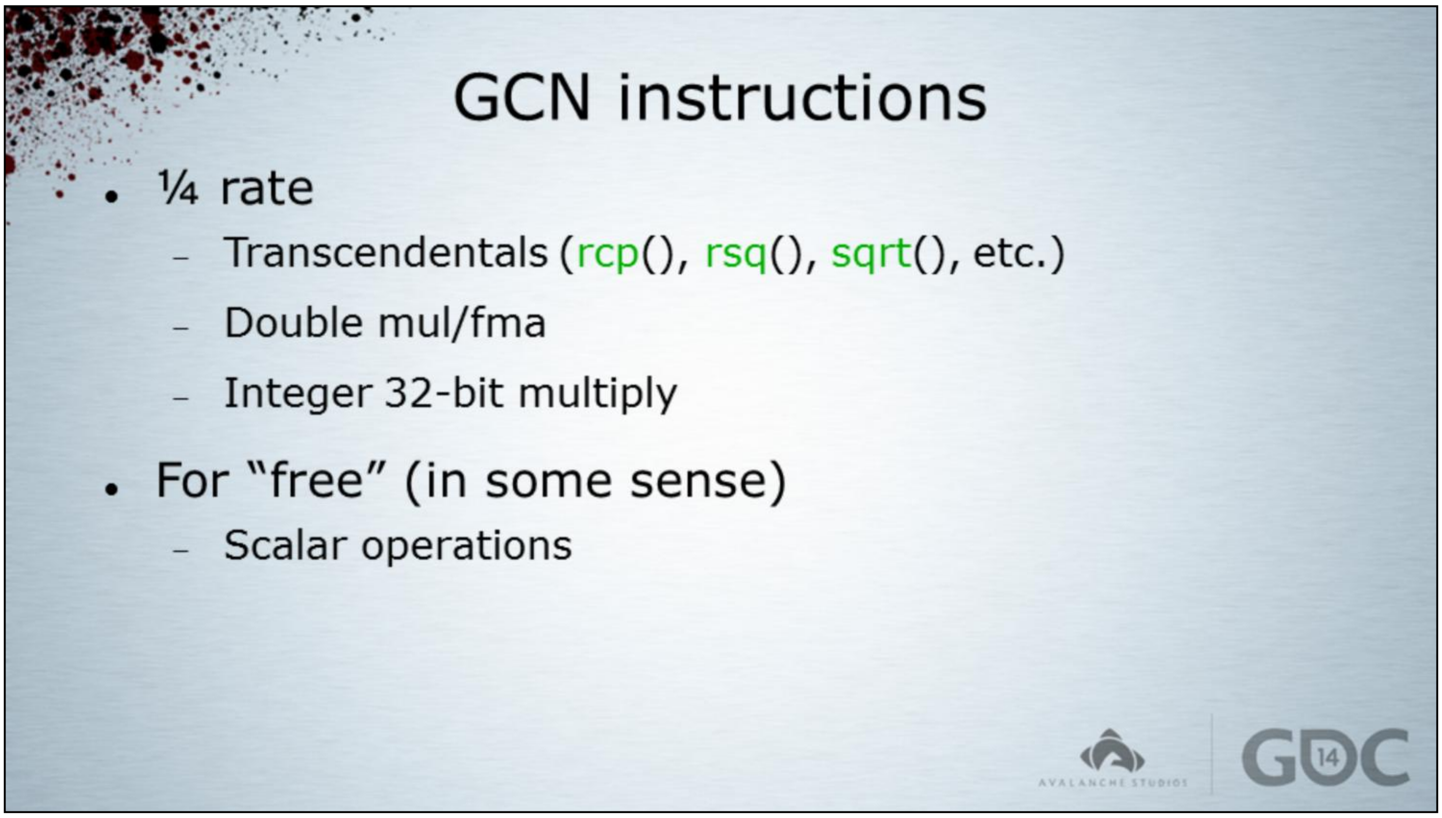
GCN instructions

- Full rate
  - Float add/sub/mul/mad/fma
  - Integer add/sub/mul24/mad24/logic
  - Type conversion, floor()/ceil()/round()
- ½ rate
  - Double add

Different instructions come at different execution speed. The full rate include your typical floating point math, as well as basic integer and logic. Note however that as far as integer multiplies goes, there is a special 24bit multiply that runs at full rate, whereas a normal 32bit multiply does not. Current hardware is still optimized primarily for floating point math, and 32bit integer multiplies are just not common enough in shaders to motivate spending the transistors to turn it into full rate. 24bit multiplies on the other hand is something you get almost for free out the floating point multiplier that the hardware already has, so that is why it can be done at full rate.

In earlier AMD hardware type conversions sat on the special transcendental unit, so you might have expected them to be quarter rate. However, in GCN they are full rate. Clamping or rounding to integers of a float are also full rate.

Double additions come at half the rate, although they also operate on twice as much data, so that is actually very good throughput.



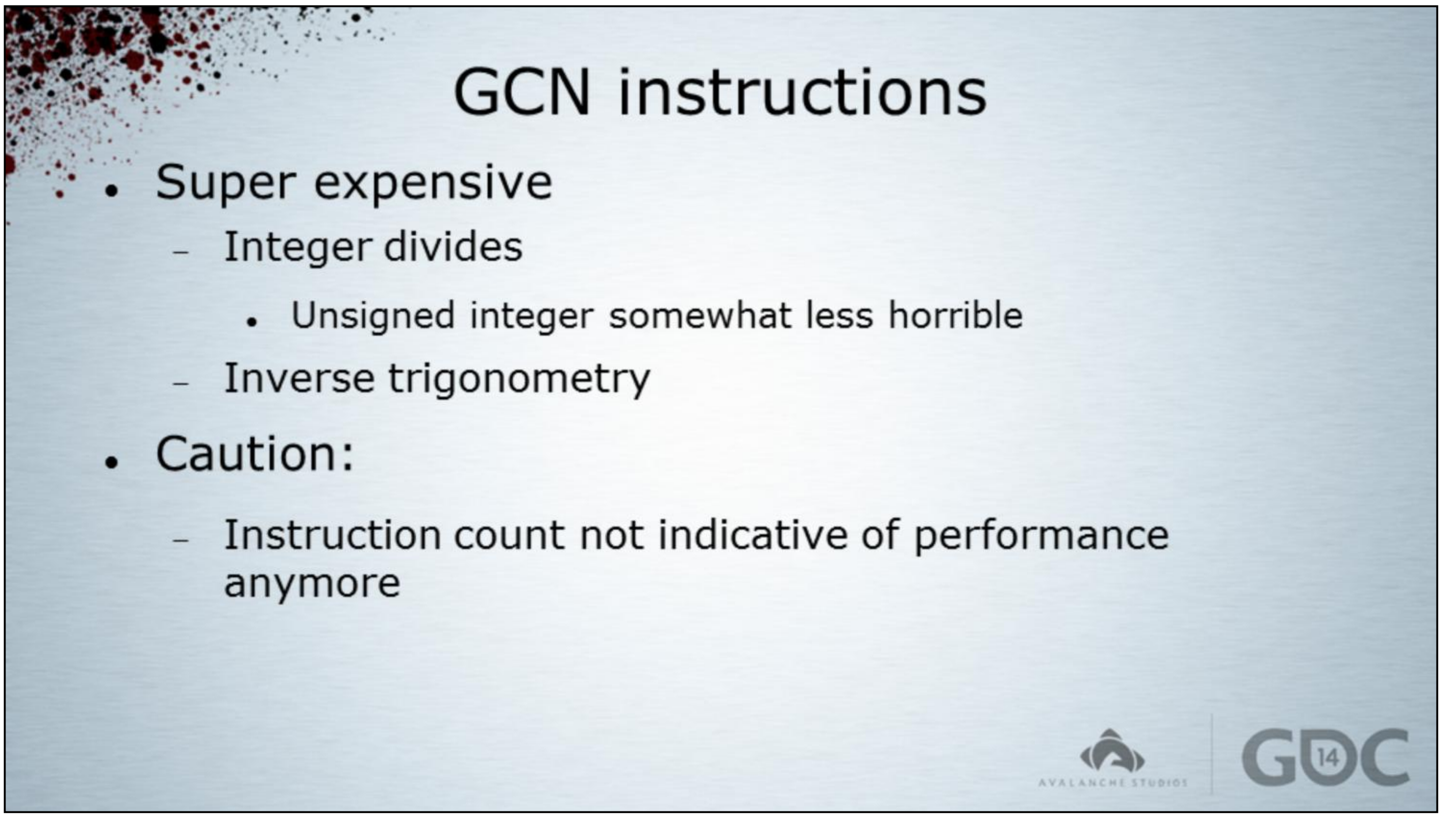
GCN instructions

- ¼ rate
  - Transcendentals (rcp(), rsq(), sqrt(), etc.)
  - Double mul/fma
  - Integer 32-bit multiply
- For "free" (in some sense)
  - Scalar operations

Transcendentals, such as square-roots, reciprocals, trigonometry, logarithms, exponentials etc. come at a quarter rate. This is also true for double multiply or the fused-multiplyadd. Note that there is no equivalent to MAD on doubles, i.e. a multiply with proper IEEE-754 rounding followed by add with proper rounding. Only a fused operation exists for doubles, with rounding in the end. This is unlikely to cause much problems, but is worth noting.

Note also that integer 32-bit multiples are quarter rate, and thus are as slow as multiplying doubles.

Scalar operations are handled by the independent scalar unit and thus does not count towards your vector instruction count. In typical shaders where math and texturing dominates, scalar operations become more or less free, although it is of course possible to make a shader be dominated by scalar instructions.

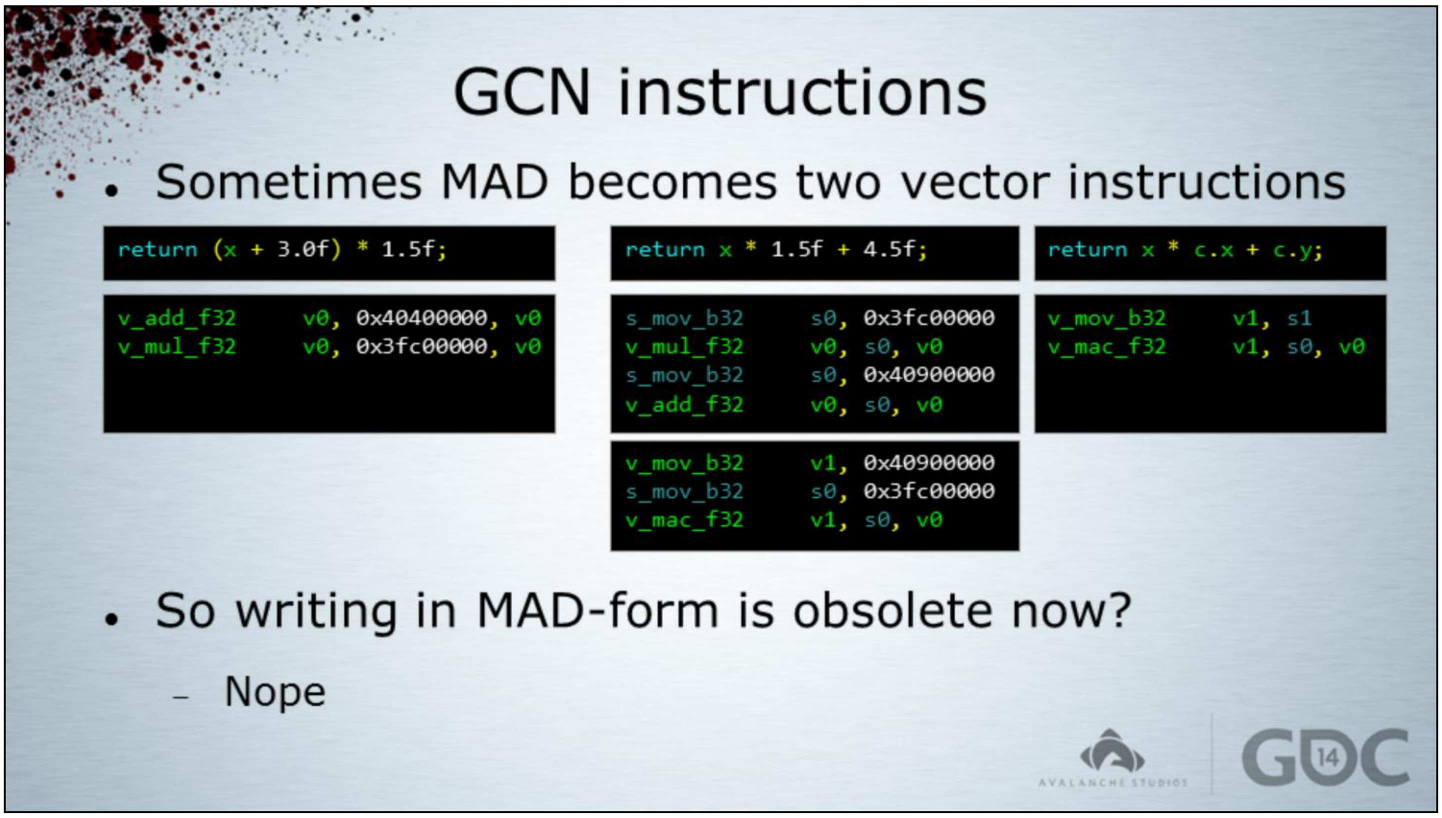


GCN instructions

- Super expensive
  - Integer divides
    - Unsigned integer somewhat less horrible
  - Inverse trigonometry
- Caution:
  - Instruction count not indicative of performance anymore

Integer division and inverse trigonometry are things you should avoid like the plague. They are not supported natively by the hardware and must be emulated with loads of instructions. The use of inverse trigonometry in particular is typically a sign in itself that you are doing something wrong. There are not a lot of cases where working with angles is the right thing to do, and almost certainly not computing them in the shader. Most likely the problem can be solved much more efficiently and elegantly using linear algebra. When reformulated things often boil down to a dot-product or two, perhaps a cross-product, and maybe a little basic math on top of that.

One thing to note is that unlike in previous AMD hardware where you could get a pretty good idea of final performance just by looking at the total VLIW slot (assuming all ALU), you cannot just look at the final instruction count on GCN and infer anything. You must look at individual instructions since their execution speed varies.



GCN instructions

- Sometimes MAD becomes two vector instructions

```
return (x + 3.0f) * 1.5f;
```

```
v_add_f32      v0, 0x40400000, v0
v_mul_f32      v0, 0x3fc00000, v0
```

```
return x * 1.5f + 4.5f;
```

```
s_mov_b32      s0, 0x3fc00000
v_mul_f32      v0, s0, v0
s_mov_b32      s0, 0x40900000
v_add_f32      v0, s0, v0
```

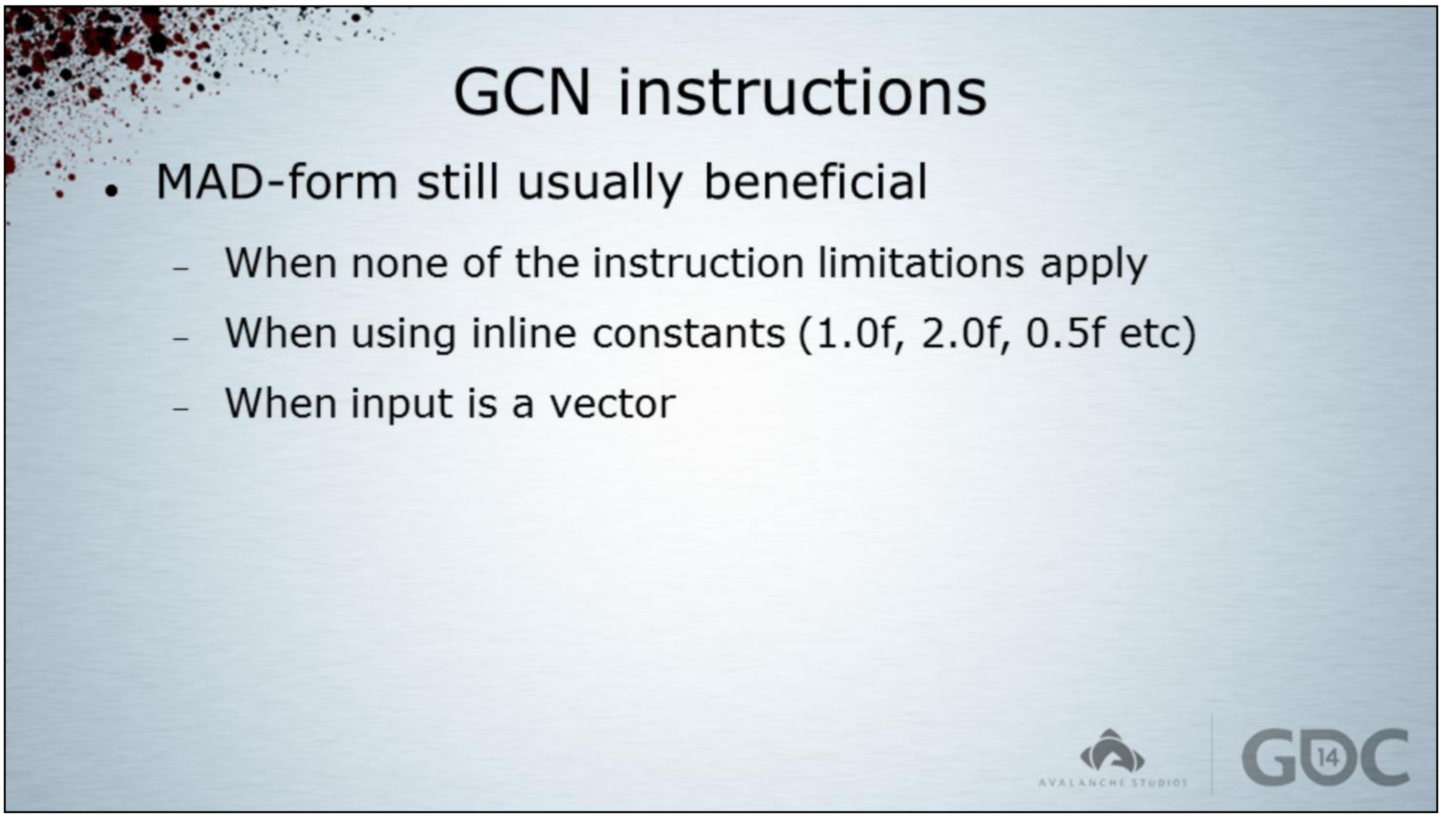
```
v_mov_b32      v1, 0x40900000
s_mov_b32      s0, 0x3fc00000
v_mac_f32      v1, s0, v0
```

```
return x * c.x + c.y;
```

```
v_mov_b32      v1, s1
v_mac_f32      v1, s0, v0
```

- So writing in MAD-form is obsolete now?
  - Nope

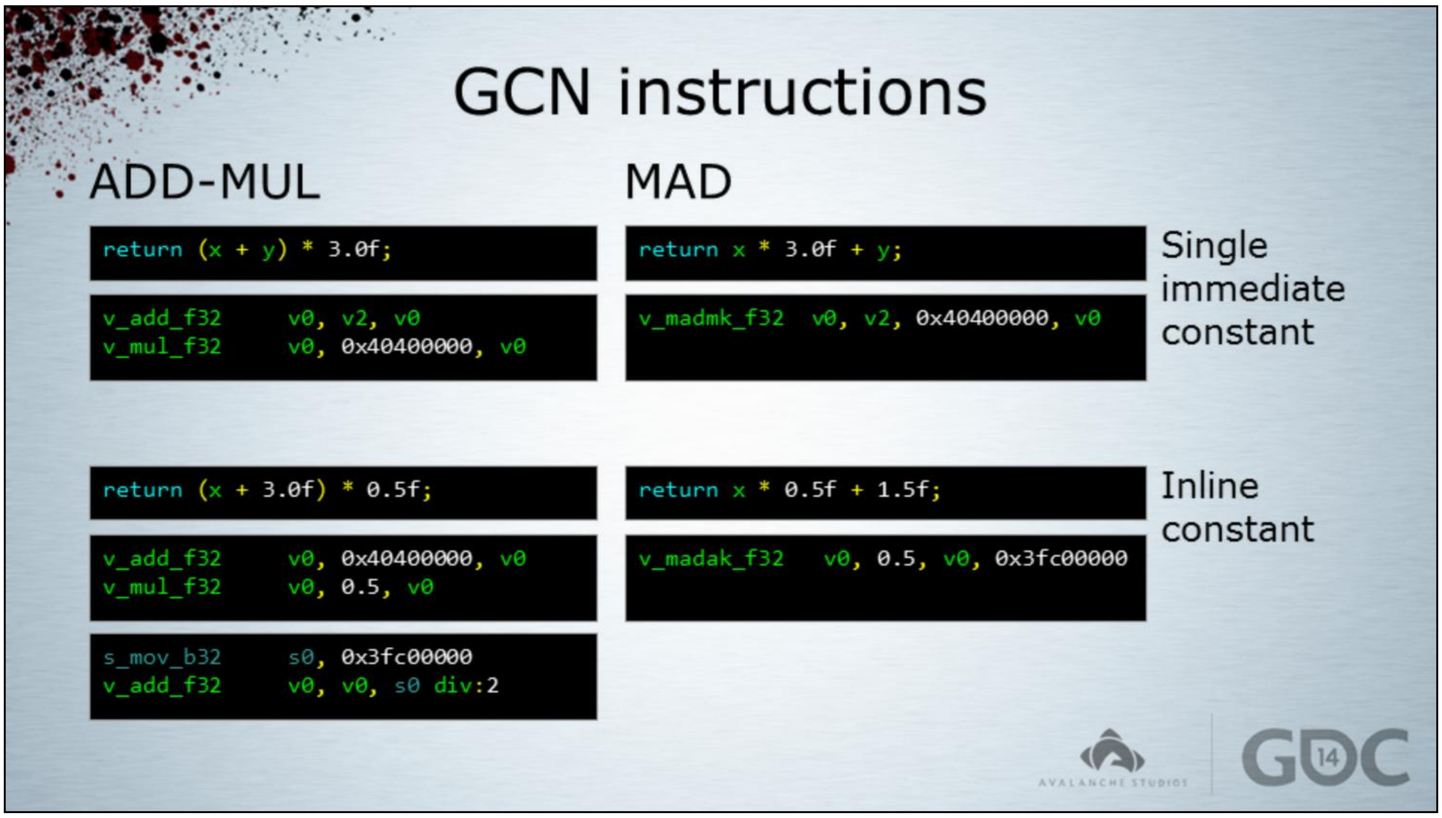
In last year's talk I put a fair amount of emphasis on the fact that all hardware since the dawn of time has been built to execute multiply-adds really fast. It has always been a single instruction on all hardware I have ever worked with, whereas an add followed by a multiply has always been two instructions. The implication thus is naturally that you should write shaders that fits this MAD-form to the greatest extent as possible. The GCN architecture complicates the issue somewhat due to its more restricted instruction set. As a general rule-of-thumb, writing in MAD-form is still the way to go; however, there are cases on GCN where it may not be beneficial, or even add a couple of scalar instructions. The shaders here illustrates a couple of failure cases. The left side represents a "before" case that consists of an add followed by multiply, which generates these two instructions as expected. The middle case uses two literal constants. These cannot both be baked into a 64bit instruction, so the compiler has to expand it to two instructions. The result differs somewhat between the platforms here, one compiler only generated a single scalar instruction and another one used two. Neither is optimal and it would have been possible to skip scalar instructions all together similar to the instruction sequence on the left, but this is what compilers at the time of this writing generated. Finally, the right side represents the case where both constants come from a constant buffer. In this case both values would lie in scalar registers, and given the read limitation of a single scalar register per instruction, one value must first be moved to a vector register before the MAD, which eliminates the benefit.



GCN instructions

- MAD-form still usually beneficial
  - When none of the instruction limitations apply
  - When using inline constants (1.0f, 2.0f, 0.5f etc)
  - When input is a vector

While there are cases where MAD-form does not result in a speedup, it is worth noting that in none of these cases does the vector instruction count increase. We are just not improving things. In the case of added scalar instructions, this is something that future compiler will hopefully address. However, the important thing to remember is that while things got a little bit worse for this in GCN, there still remains plenty of cases where writing in MAD-form is still beneficial.



GCN instructions

ADD-MUL

```
return (x + y) * 3.0f;
```

```
v_add_f32     v0, v2, v0
v_mul_f32     v0, 0x40400000, v0
```

```
return (x + 3.0f) * 0.5f;
```

```
v_add_f32     v0, 0x40400000, v0
v_mul_f32     v0, 0.5, v0
```

```
s_mov_b32     s0, 0x3fc00000
v_add_f32     v0, v0, s0 div:2
```

MAD

```
return x * 3.0f + y;
```

```
v_madmk_f32  v0, v2, 0x40400000, v0
```

Single immediate constant

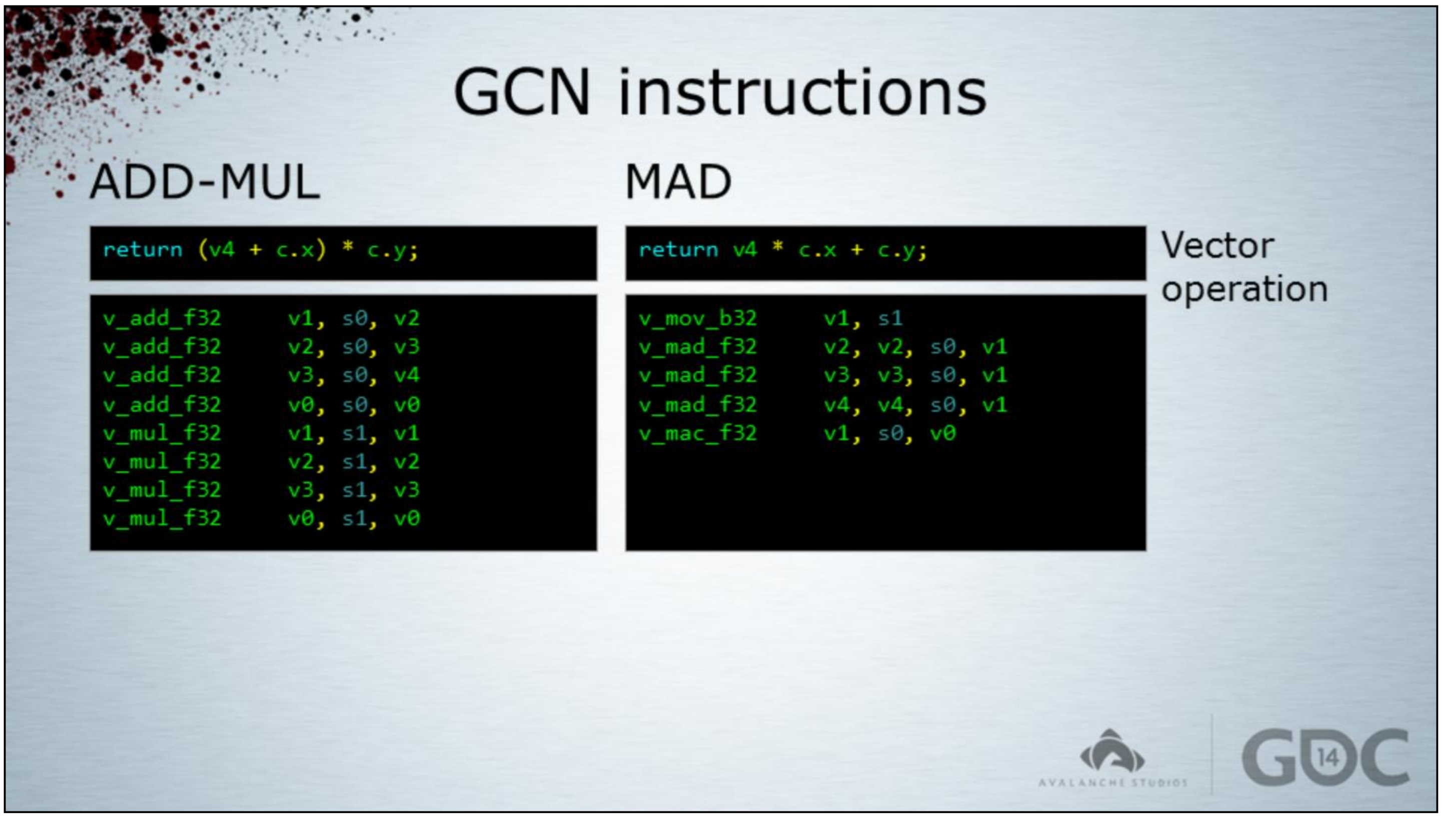
```
return x * 0.5f + 1.5f;
```

```
v_madak_f32   v0, 0.5, v0, 0x3fc00000
```

Inline constant

This illustrates two cases where writing in MAD-form is still beneficial on GCN. The first case only needs a single immediate constant, so this case works. Note that the left and right panels are not equivalent code, but just illustrates the difference between these forms.

The other example shows how using one of the special inline constants also fixes it. There these two examples are equivalent and the MAD-form runs faster. It is worth noting though that one compiler was able to optimize the ADD-MUL case using the output multiplier instead, resulting in a single vector instruction even for that case.



GCN instructions

ADD-MUL

```
return (v4 + c.x) * c.y;
```

```
v_add_f32     v1, s0, v2
v_add_f32     v2, s0, v3
v_add_f32     v3, s0, v4
v_add_f32     v0, s0, v0
v_mul_f32     v1, s1, v1
v_mul_f32     v2, s1, v2
v_mul_f32     v3, s1, v3
v_mul_f32     v0, s1, v0
```

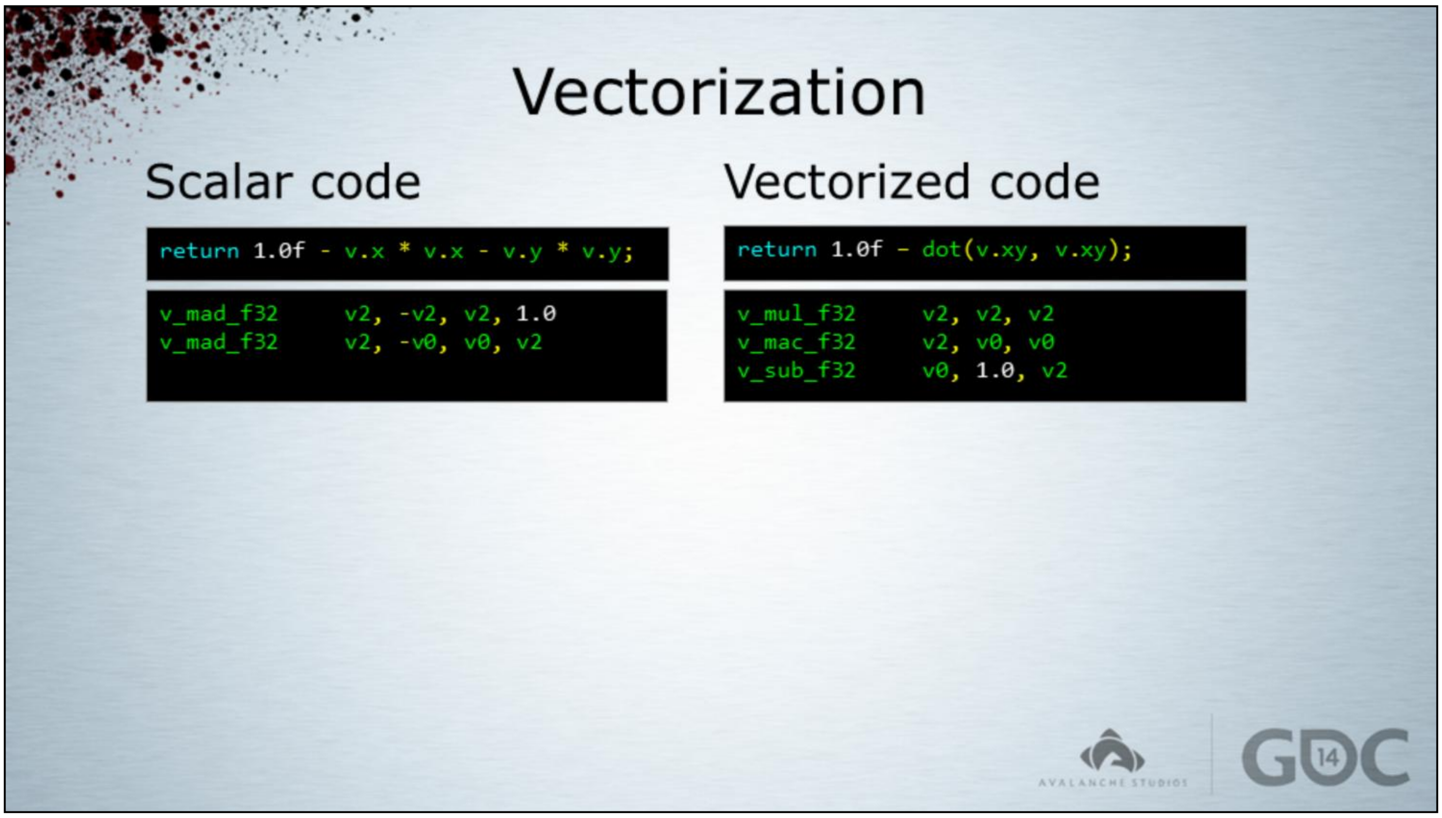
MAD

```
return v4 * c.x + c.y;
```

```
v_mov_b32     v1, s1
v_mad_f32     v2, v2, s0, v1
v_mad_f32     v3, v3, s0, v1
v_mad_f32     v4, v4, s0, v1
v_mac_f32     v1, s0, v0
```

Vector operation

Finally, when using a vector, the extra overhead from GCN limitations can be amortized and still result in a substantial overall improvement. Instead of cutting down this float4 add-multiply from 8 to 4 operations as on previous hardware, we only go from 8 to 5, but that is still a good improvement. While we would not see a benefit on a single float in MAD-form here, even with a two-component vector would begin to see an improvement.



Vectorization

Scalar code

```
return 1.0f - v.x * v.x - v.y * v.y;
```

```
v_mad_f32     v2, -v2, v2, 1.0
v_mad_f32     v2, -v0, v0, v2
```

Vectorized code

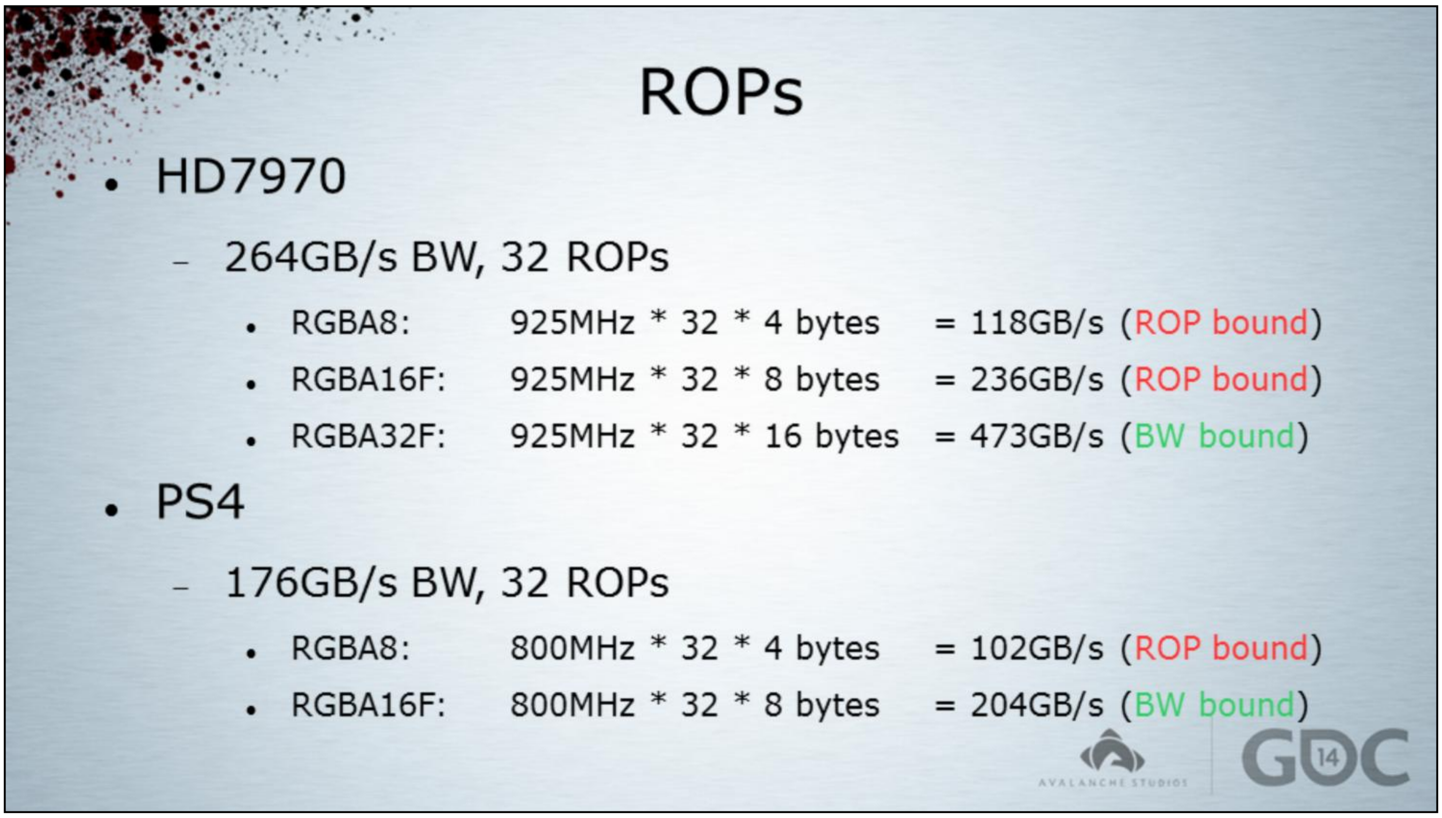
```
return 1.0f - dot(v.xy, v.xy);
```

```
v_mul_f32     v2, v2, v2
v_mac_f32     v2, v0, v0
v_sub_f32     v0, 1.0, v2
```

In the past people often tried to write vectorized code in order to better take advantage of the underlying vector architectures. People bunched together separate things into vectors and did math on vectors instead of writing things in a more intuitive form. This made sense in 2005 perhaps, when GPUs were vector based and shader optimizers were not particularly sophisticated. However, since DX10 GPUs arrived everything has been scalar or VLIW, making explicit vectorization dubious at best and counter-productive at worst. If explicit vectorization introduces extra math operations, it will only slow things down. Refer to last year's talk and the topic of separating scalar and vector work to see how that could easily happen.

Fortunately I do not see much explicit vectorization these days, the remaining exception may be trying to use dot-products in hope of taking advantage of a build-in instruction. This may work on vector-based architectures, but scalar architectures do not even have a dot-product instruction. Instead it is implemented as a series of MUL and MADs. Even on VLIW architectures that do have a dot-product instruction it is unlikely to speed things up since you will occupy at least as many lanes and possibly more.

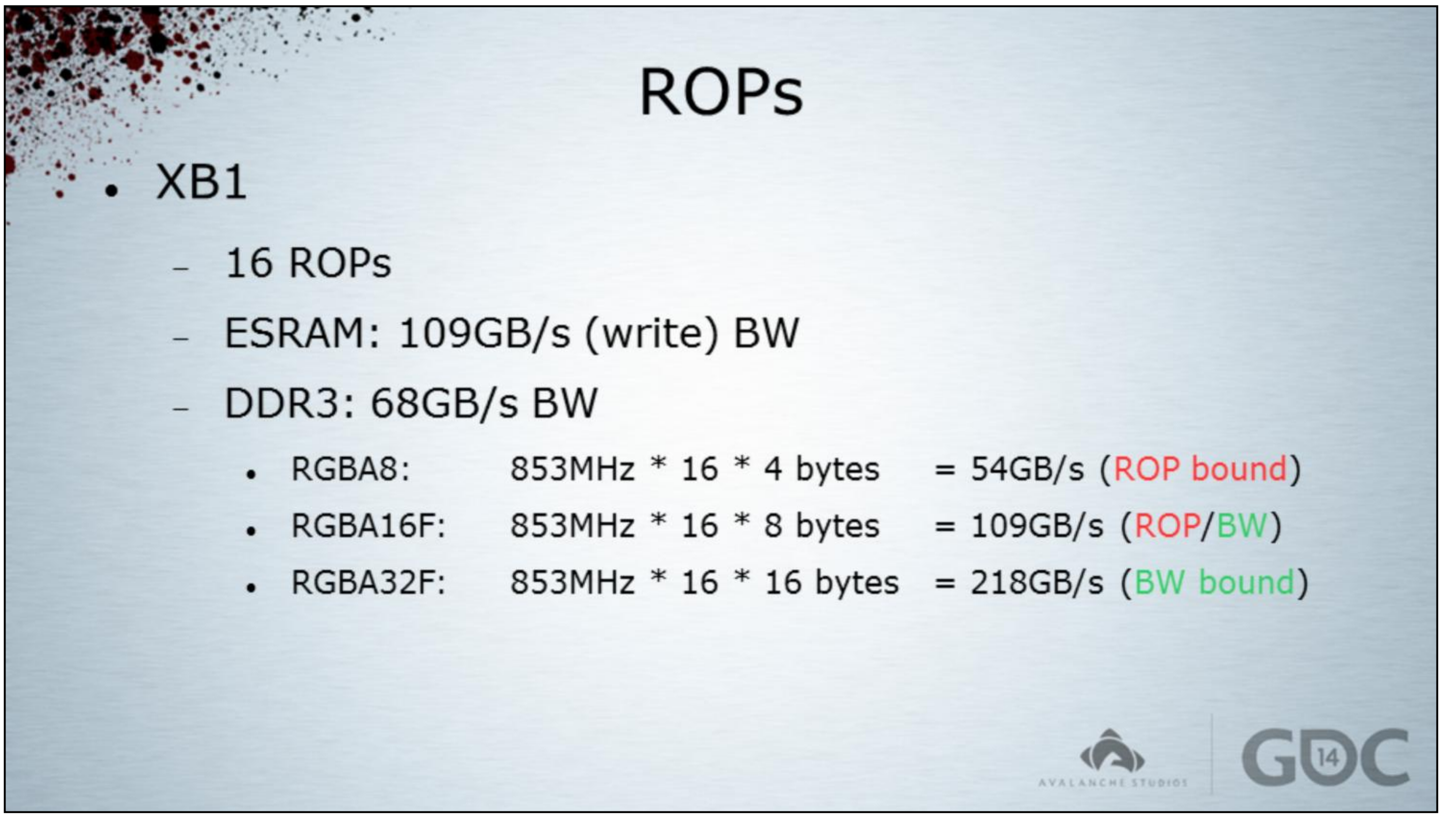
The problem in the example here is that the evaluation order will require that the dot-product is evaluated first, followed by the subtraction. This unnecessarily prevents the compiler from merging the subtraction and multiplication into a single MAD. This is not the case for the expression on the left.



ROPs

- HD7970
  - 264GB/s BW, 32 ROPs
    - RGBA8: 925MHz * 32 * 4 bytes = 118GB/s (ROP bound)
    - RGBA16F: 925MHz * 32 * 8 bytes = 236GB/s (ROP bound)
    - RGBA32F: 925MHz * 32 * 16 bytes = 473GB/s (BW bound)
- PS4
  - 176GB/s BW, 32 ROPs
    - RGBA8: 800MHz * 32 * 4 bytes = 102GB/s (ROP bound)
    - RGBA16F: 800MHz * 32 * 8 bytes = 204GB/s (BW bound)

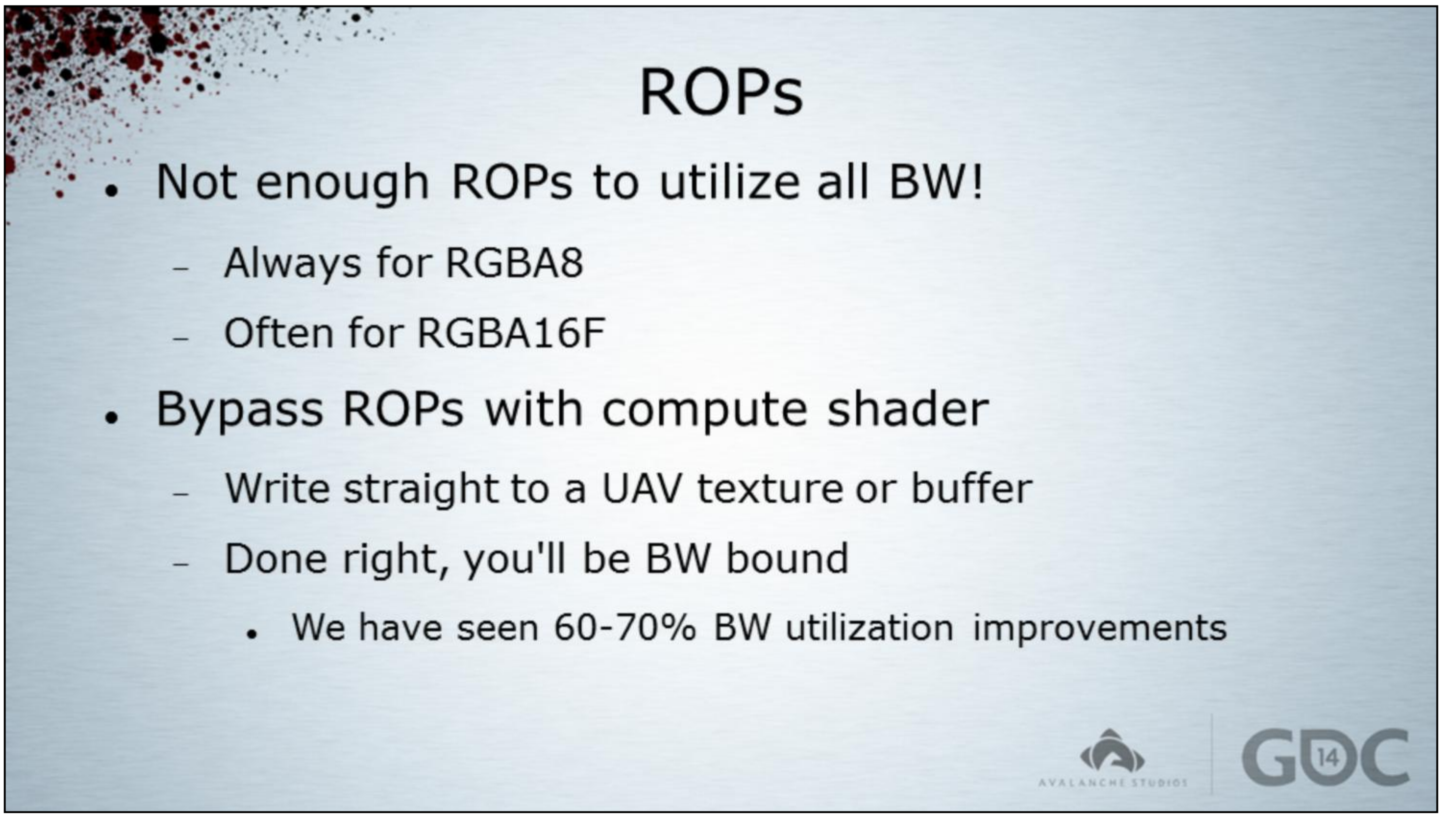
As hardware has gotten increasingly more powerful over the years, some parts of it has lagged behind. The number of ROPs (i.e. how many pixels we can output per clock) remains very low. While this reflects typical use cases where the shader is reasonably long, it may limit the performance of short shaders. Unless the output format is wide, we are not even theoretically capable of using the full bandwidth available. For the HD7970 we need a 128bit format to become bandwidth bound. For the PS4 64bit would suffice.



ROPs

- XB1
  - 16 ROPs
  - ESRAM: 109GB/s (write) BW
  - DDR3: 68GB/s BW
    - RGBA8: 853MHz * 16 * 4 bytes = 54GB/s (ROP bound)
    - RGBA16F: 853MHz * 16 * 8 bytes = 109GB/s (ROP/BW)
    - RGBA32F: 853MHz * 16 * 16 bytes = 218GB/s (BW bound)

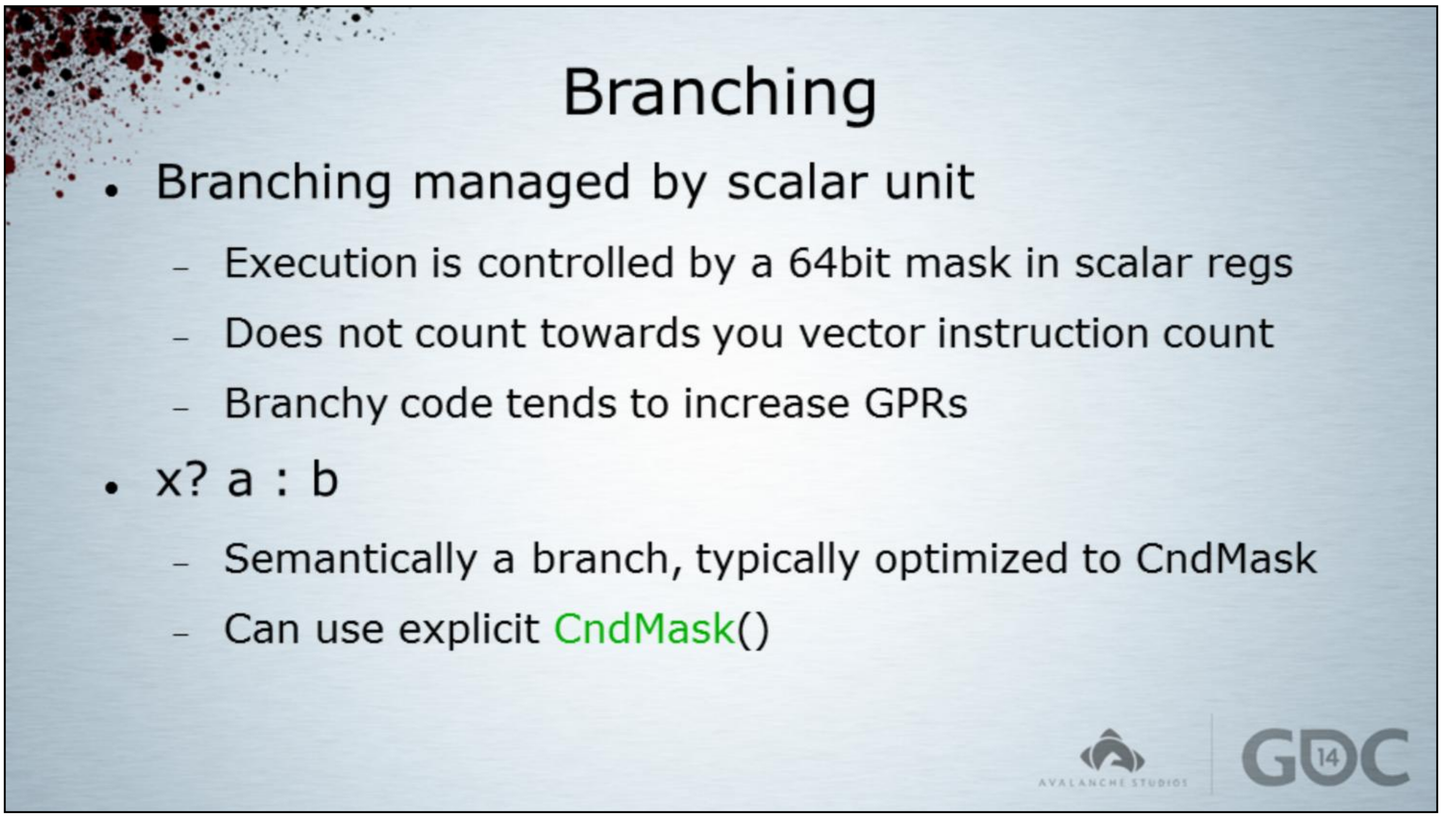
On the XB1, if we are rendering to ESRAM, 64bit just about hits the crossover point between ROP and bandwidth-bound. But even if we render to the relatively slow DDR3 memory, we will still be ROP-bound if the render-target is a typical 32bit texture.



ROPs

- Not enough ROPs to utilize all BW!
  - Always for RGBA8
  - Often for RGBA16F
- Bypass ROPs with compute shader
  - Write straight to a UAV texture or buffer
  - Done right, you'll be BW bound
    - We have seen 60-70% BW utilization improvements

The solution is to use a compute shader. Writing through a UAV bypasses the ROPs and goes straight to memory. This solution obviously does not apply to all sorts of rendering, for one we are skipping the entire graphics pipeline as well on which we still depend for most normal rendering. However, in the cases where it applies it can certainly result in a substantial performance increase. Cases where we are initializing textures to something else than a constant color, simple post-effects, this would be useful.

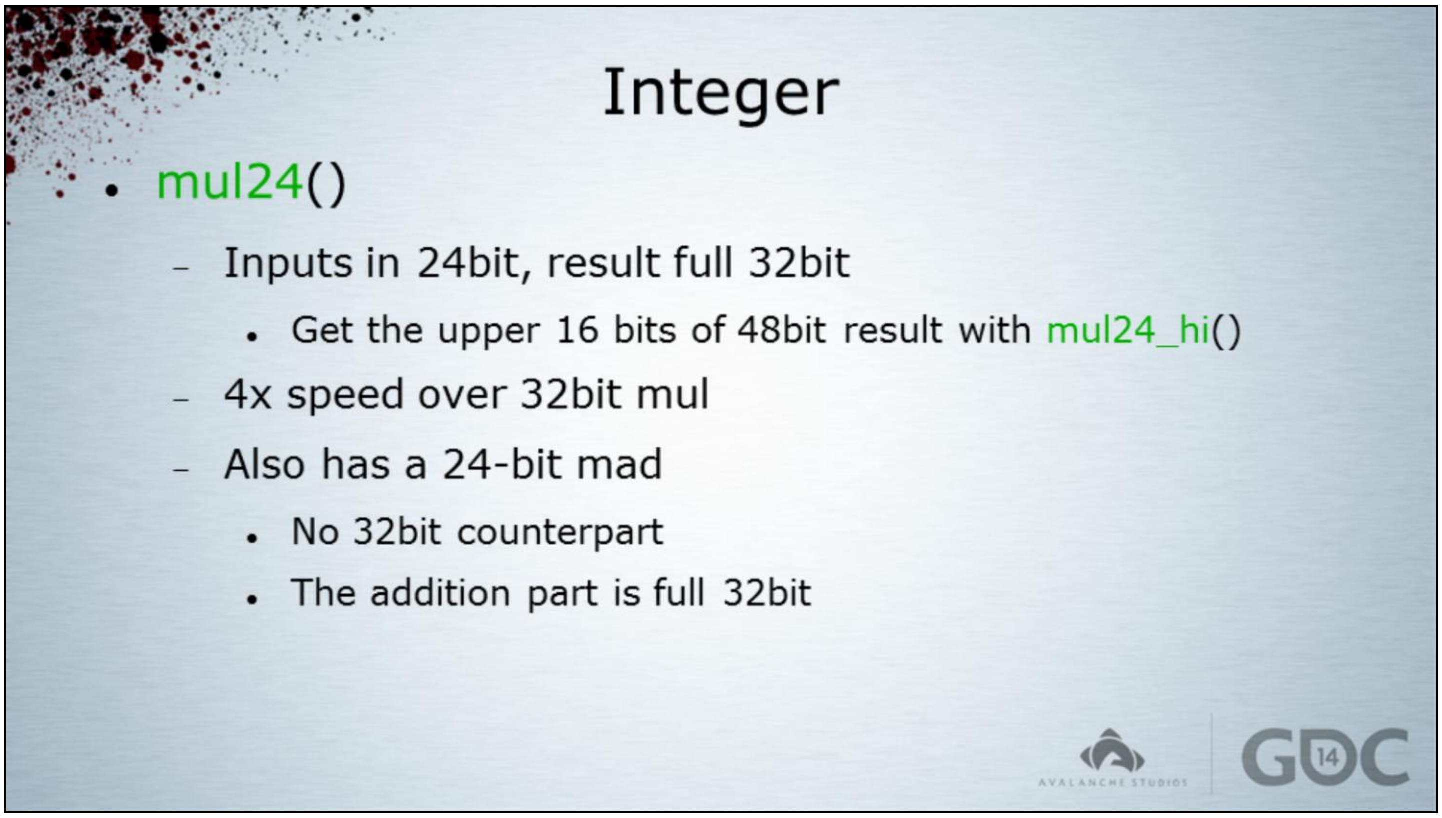


Branching

- Branching managed by scalar unit
  - Execution is controlled by a 64bit mask in scalar regs
  - Does not count towards you vector instruction count
  - Branchy code tends to increase GPRs
- x? a : b
  - Semantically a branch, typically optimized to CndMask
  - Can use explicit CndMask()

Branching on GCN is generally speaking very fast, provided of course that we have good coherency and so on. The overhead from the branching itself is very low. Other than doing the comparisons that we are branching on, it is more or less free given that the actual branching is handled by the scalar unit. The thing to look out for, however, is the register pressure. Very branchy code tends to increase the number of registers the shader requires. This can impact performance much more significantly than the branching itself does.

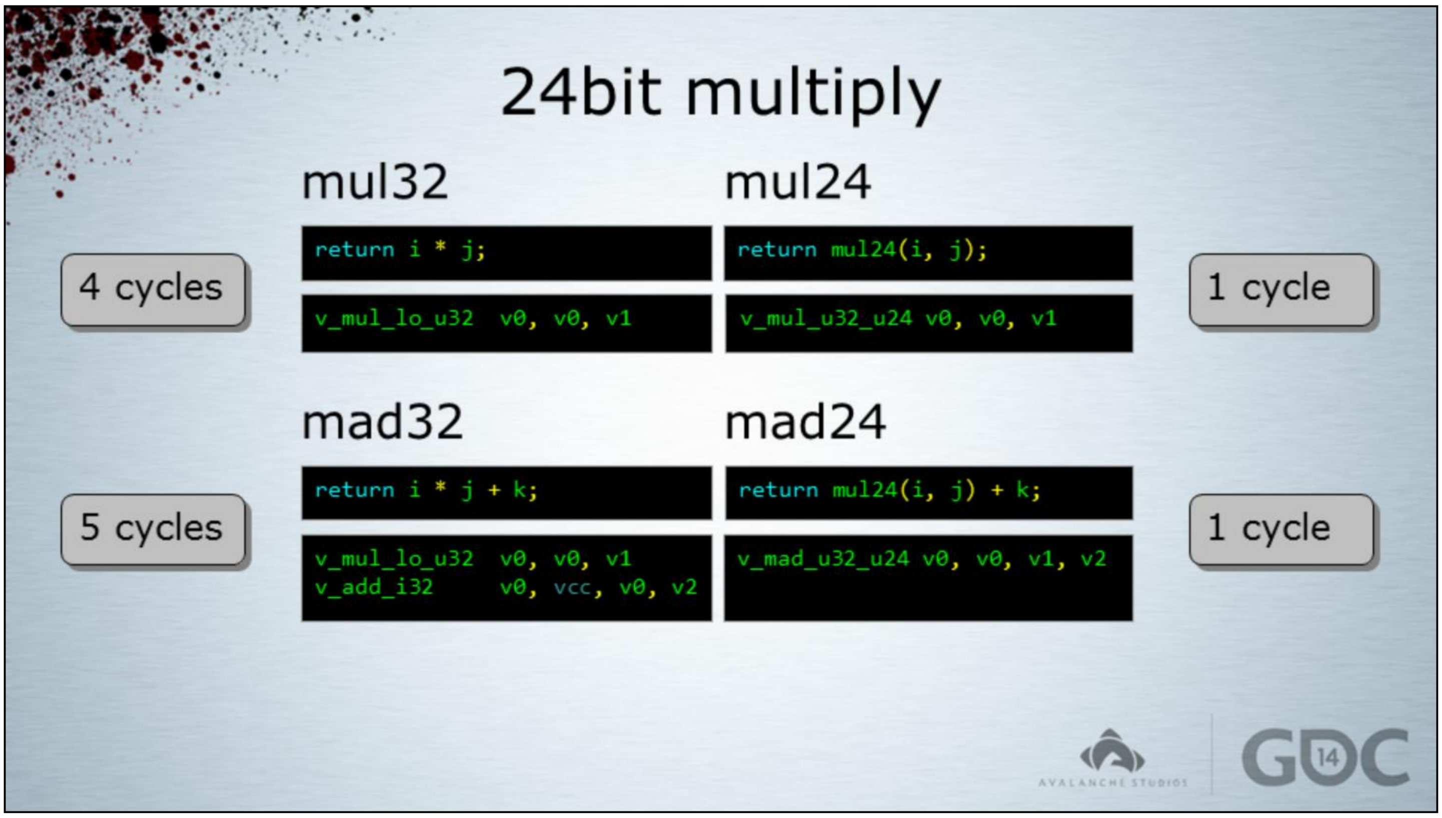
Most of the time branches are fine. For very tiny branches, a conditional assignment may be preferable. The easiest way to do that is to simply use the ?-operator. Semantically it is a branch though, rather than a conditional assignment, although simple expressions written that way tends to compile down to a conditional assignment. However, we have observed cases on some platforms where this didn’t happen and a full branch was emitted. In this case it may help to use an explicit CndMask() that maps straight to the underlying conditional mask instruction.



Integer

- mul24()
  - Inputs in 24bit, result full 32bit
    - Get the upper 16 bits of 48bit result with mul24_hi()
  - 4x speed over 32bit mul
  - Also has a 24-bit mad
    - No 32bit counterpart
    - The addition part is full 32bit

As discussed earlier, 24bit multiplies are full rate whereas 32bit multiplies are quarter rate, so the 24bit is preferable in the multitude of cases where it is sufficient. It is worth noting that while the inputs are 24bit, the result is a full 32bit value. In fact, the hardware can also give you the top 16 bits from the 48bit result, should you have a use for it. There is also a MAD version of the 24bit multiply (unlike the 32bit one) and it is worth pointing out that for the addition part the input is the full 32 bits, not 24.



24bit multiply

mul32

4 cycles

```
return i * j;
```

```
v_mul_lo_u32  v0, v0, v1
```

mul24

1 cycle

```
return mul24(i, j);
```

```
v_mul_u32_u24 v0, v0, v1
```

mad32

5 cycles

```
return i * j + k;
```

```
v_mul_lo_u32  v0, v0, v1
v_add_i32     v0, vcc, v0, v2
```

mad24

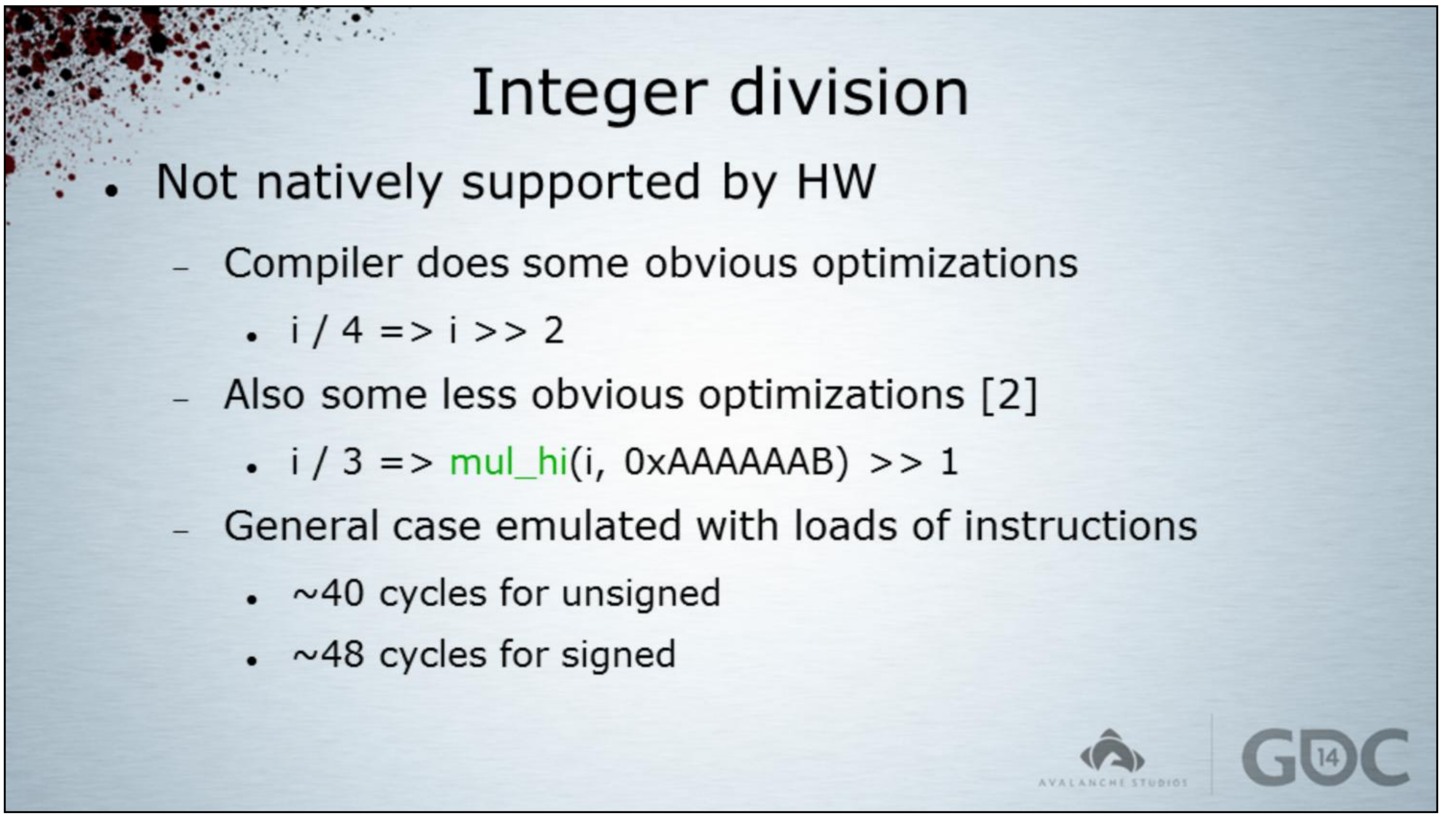
1 cycle

```
return mul24(i, j) + k;
```

```
v_mad_u32_u24 v0, v0, v1, v2
```

AVALANCHE STUDIOS
GDC 14

As mentioned, 24bit multiply is full rate whereas 32bit is quarter rate. The difference is further expanded when you also need to add as well. The addition can be merged into a 24bit MAD, whereas no such instruction exists for 32bit multiplies. This makes 32bit multiply-add 5 times slower than 24bit.

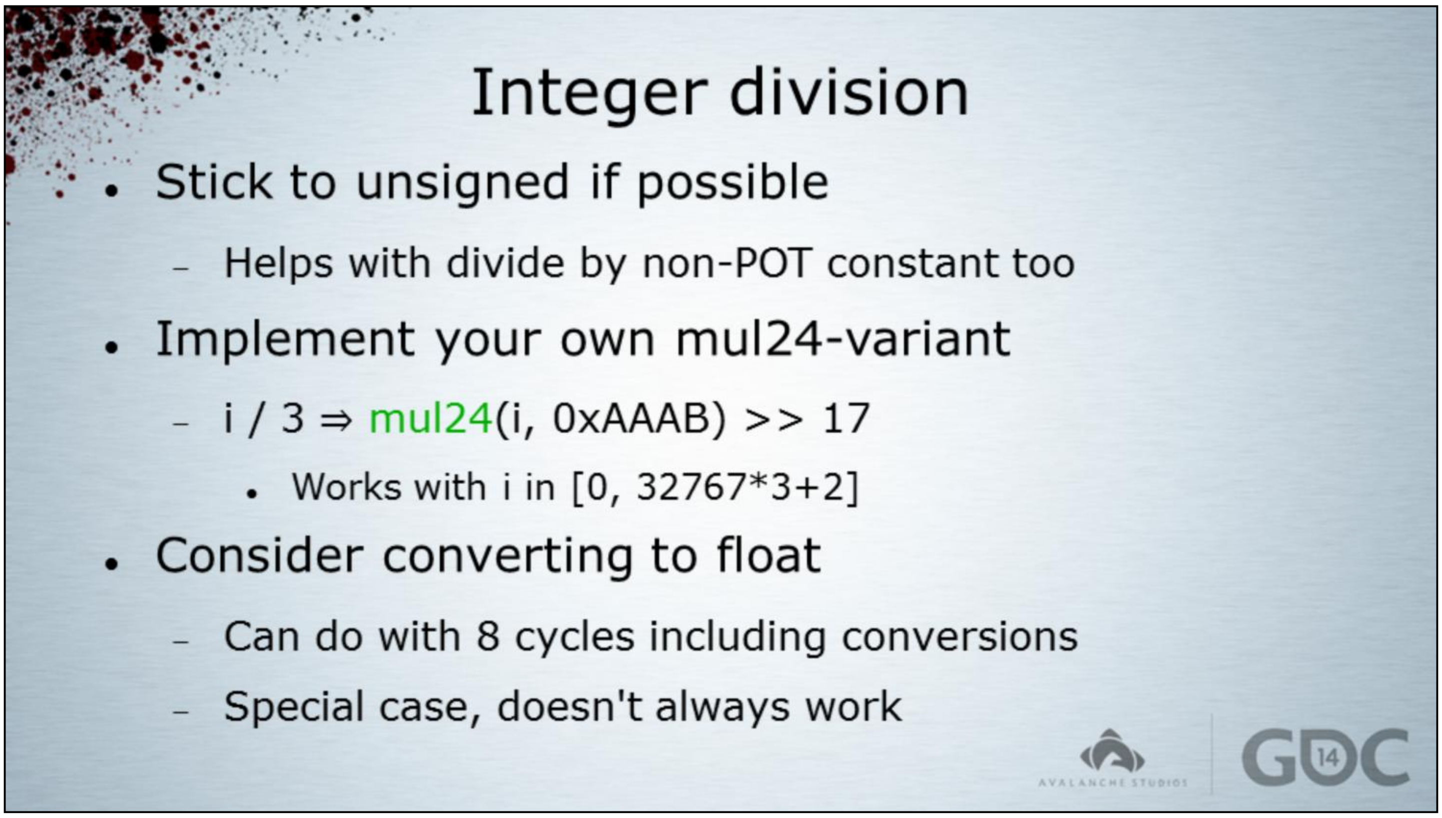


Integer division

- Not natively supported by HW
  - Compiler does some obvious optimizations
    - i / 4 => i >> 2
  - Also some less obvious optimizations [2]
    - i / 3 => mul_hi(i, 0xAAAAAAB) >> 1
  - General case emulated with loads of instructions
    - ~40 cycles for unsigned
    - ~48 cycles for signed

The hardware does not support any form of integer division natively. For some obvious cases, like dividing by a power-of-two constant the compiler does the obvious conversion of a bit shift. For the more generic division by constant there are known methods for generating a short sequence of a multiply by magic constant and some shifts and add/sub. The exact sequence depends on your input, so the cost is variable. The expensive part is the multiplication by the magic number since it is done in full 32bit, followed by 1-4 basic ALUs, for a total cost of 5-8 cycles. For signed integer division it is more expensive.

For the general case where the denominator is unknown the division is emulated with loads of instruction. This should obviously be avoided if at all possible.



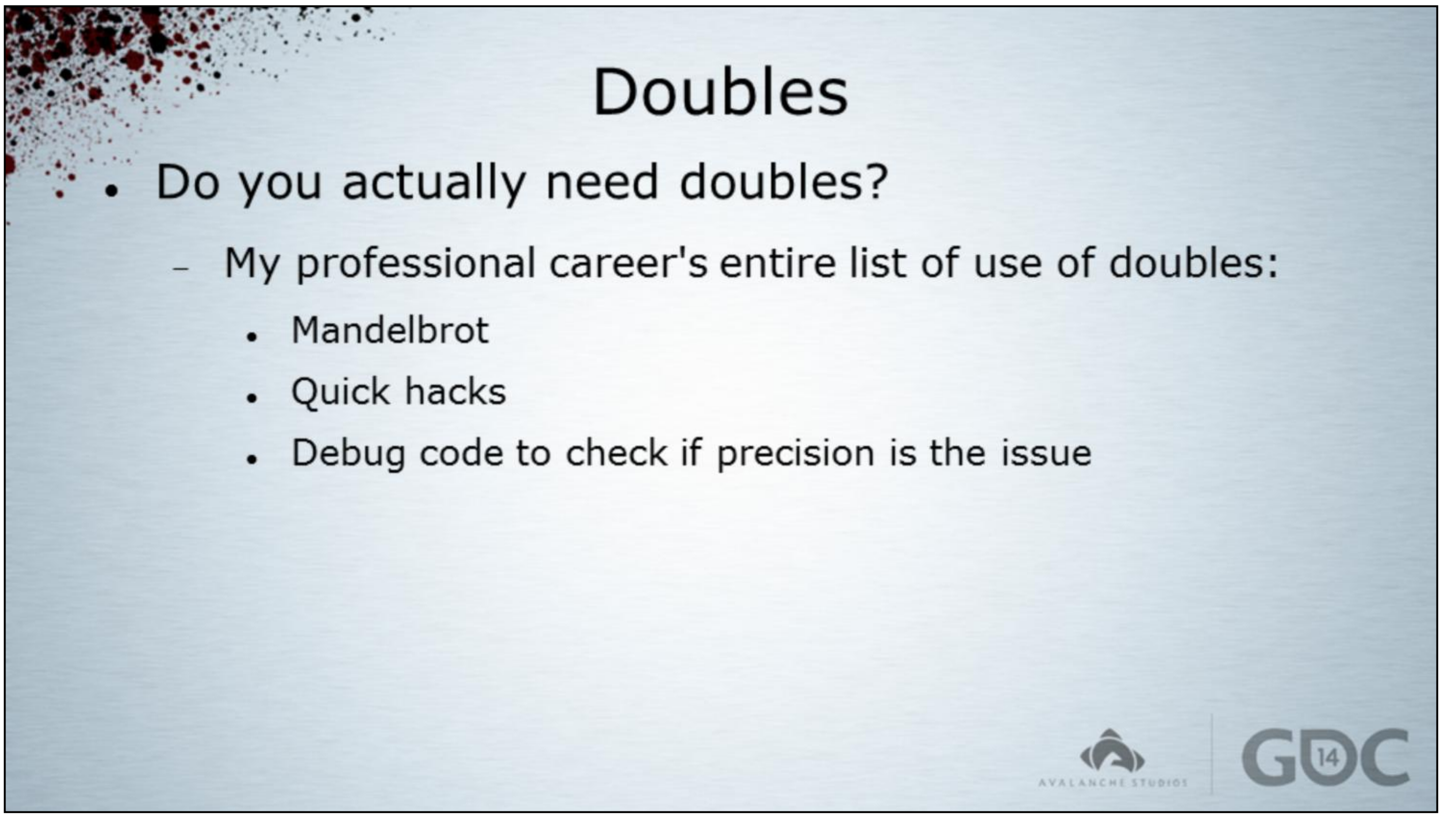
Integer division

- Stick to unsigned if possible
  - Helps with divide by non-POT constant too
- Implement your own mul24-variant
  - i / 3 ⇒ mul24(i, 0xAAAB) >> 17
    - Works with i in [0, 32767*3+2]
- Consider converting to float
  - Can do with 8 cycles including conversions
  - Special case, doesn't always work

The easiest optimization is to simply switch to unsigned if you do not need to deal with negative numbers. This has a large impact even on division by constant.

If you are dividing by constant, one thing you can do if your inputs are limited enough in range is to implement your own mul24 based version. For instance a division by 3 can be done in 2 cycles instead of the 5 required for a full 32bit. This works as long as the input is no larger than 98303.

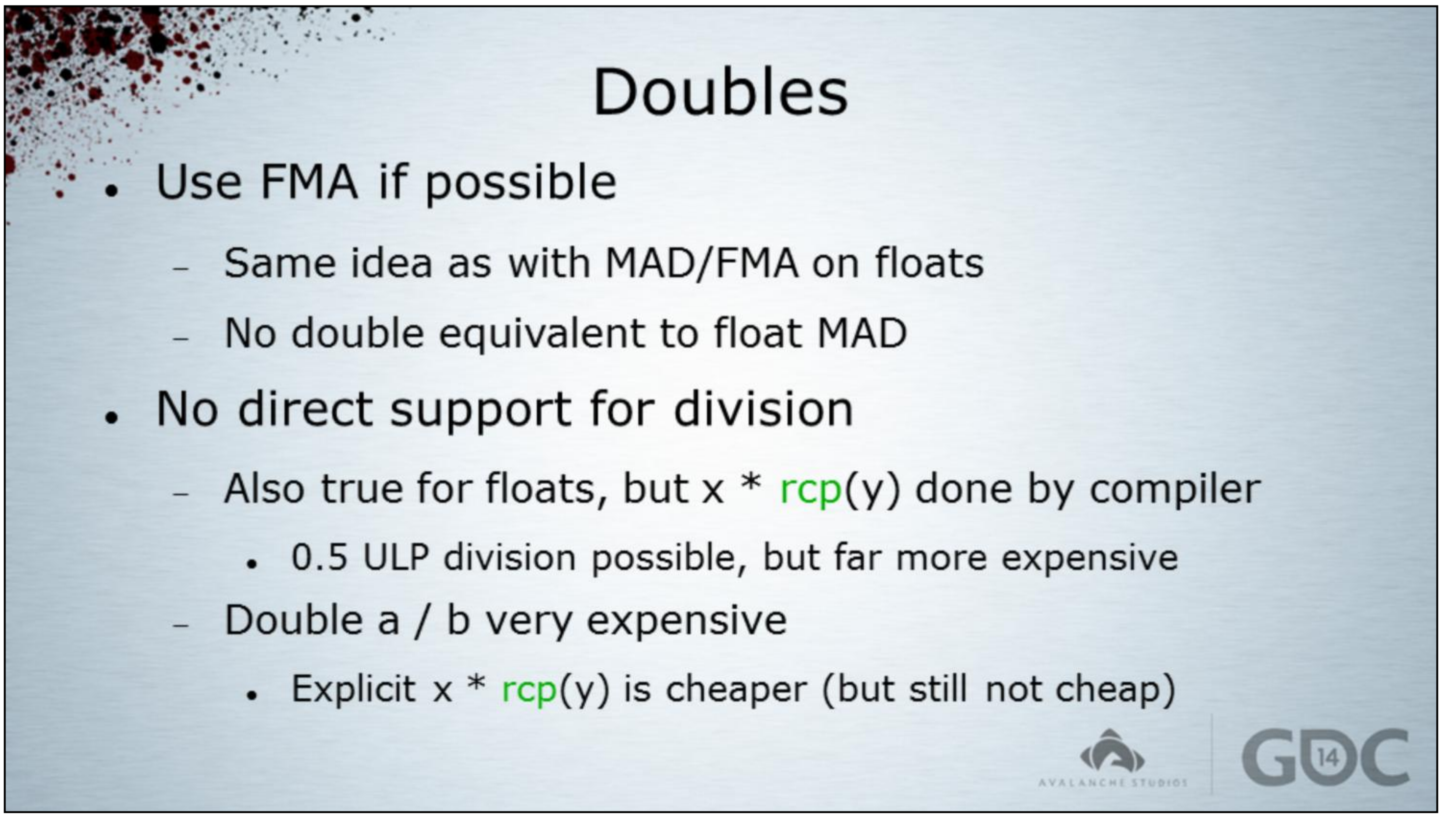
For some specific cases it may be possible to do the division in floating point instead. Including conversions back and forth it will be 8 cycles. You may have to take special case to handle precision and rounding here, so be careful and do proper testing if you go down this path.



Doubles

- Do you actually need doubles?
  - My professional career's entire list of use of doubles:
    - Mandelbrot
    - Quick hacks
    - Debug code to check if precision is the issue

GCN supports doubles. First question if you consider using it is if you actually need them. There is a good chance that you do not. In my entire career doubles have been the answer (aside from quick hacks and tests) pretty much only in Mandelbrot rendering, and in that case it is mostly just the better choice compared to floats rather than the final solution, ideally I would like even more precision. However, there are certainly scientific computations and other applications where the use of doubles is a sensible choice, and in that case there are a few things you can do.



Doubles

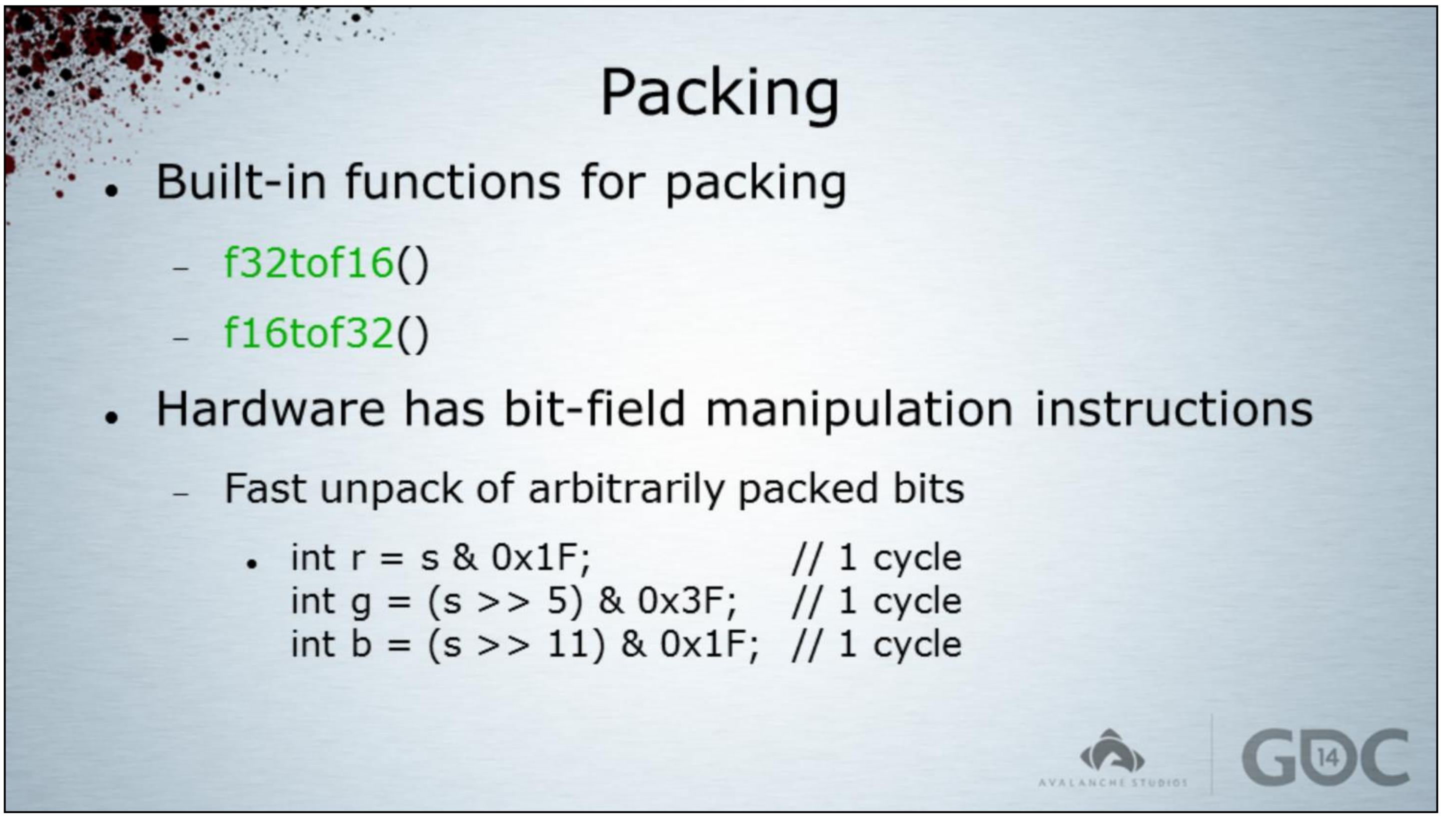
- Use FMA if possible
  - Same idea as with MAD/FMA on floats
  - No double equivalent to float MAD
- No direct support for division
  - Also true for floats, but x * rcp(y) done by compiler
    - 0.5 ULP division possible, but far more expensive
  - Double a / b very expensive
    - Explicit x * rcp(y) is cheaper (but still not cheap)

AVALANCHE STUDIOS
GDC 14

Much like with floats, doubles benefit from MAD-form. Note that there is no IEEE-754 compatible MAD instruction for doubles that rounds between the multiply and add. There is only a fused multiply-add, with a single rounding in the end. I would imagine that this would rarely cause any problems for any typical applications, and FMA is certainly the preferable choice if you are not constrained by strict IEEE-754 compatibility.

There is no direct support for double division in GCN. Actually not for floats either. For floats the compiler simply accept 1.0 ULP, thus allowing it to be implemented as an rcp followed by a multiply. However, the hardware has some supporting functions for the case where you really really need that proper 0.5 ULP result, allowing you to potentially flip the last bit on your float at the cost of lots of cycles.

Note that for doubles the compiler implements a strict 0.5 ULP by default. This means you will go down the expensive path by default, unlike for floats. If this is not desired and you can live with somewhat worse precision in your divisions, you can implement the division as an rcp and multiply manually. This will reduce the cost somewhat, although it is still not cheap by any means.

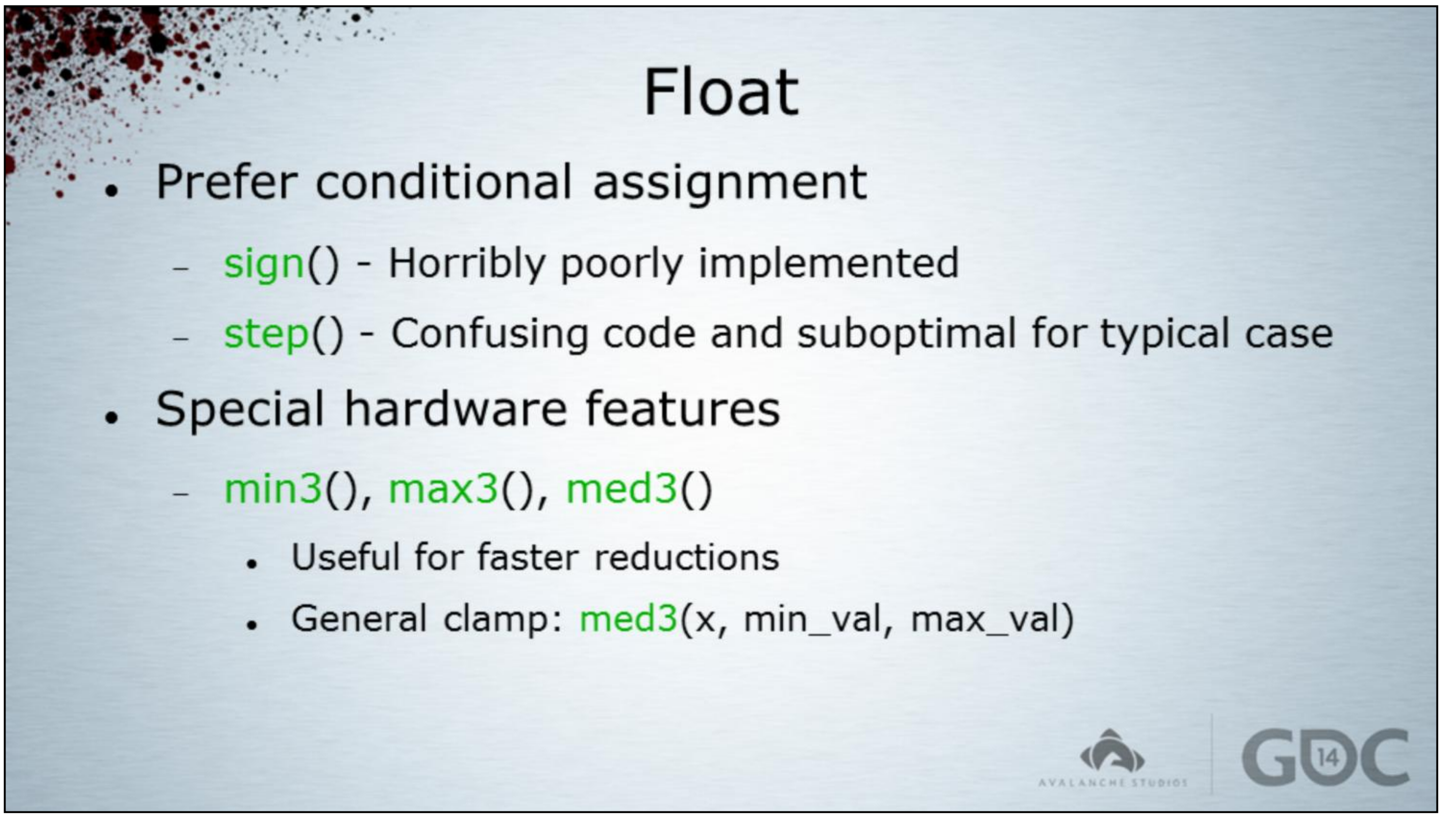


Packing

- Built-in functions for packing
  - f32tof16()
  - f16tof32()
- Hardware has bit-field manipulation instructions
  - Fast unpack of arbitrarily packed bits

    -
      ```
      int r = s & 0x1F;             // 1 cycle
      int g = (s >> 5) & 0x3F;      // 1 cycle
      int b = (s >> 11) & 0x1F;     // 1 cycle
      ```

With every generation the ALU power keeps increasing at a far greater rate than the available bandwidth. As such, it is of increasing important to pack your data as tightly as possible. There are a few handy features for fast pack and unpack. Firstly the standard DX11 functions for converting floats to half and vice versa. The other is the bit-field manipulation instructions on GCN. These are exposed as intrinsics on some platforms and can be used explicitly there. In other cases the compiler will typically spot extraction and packing of bit-fields and generate these instructions for you. The case illustrated here with an unpacking of a good old R5G6B5 color will compile into only three instructions, rather than five that you might expect with standard bitwise operations.

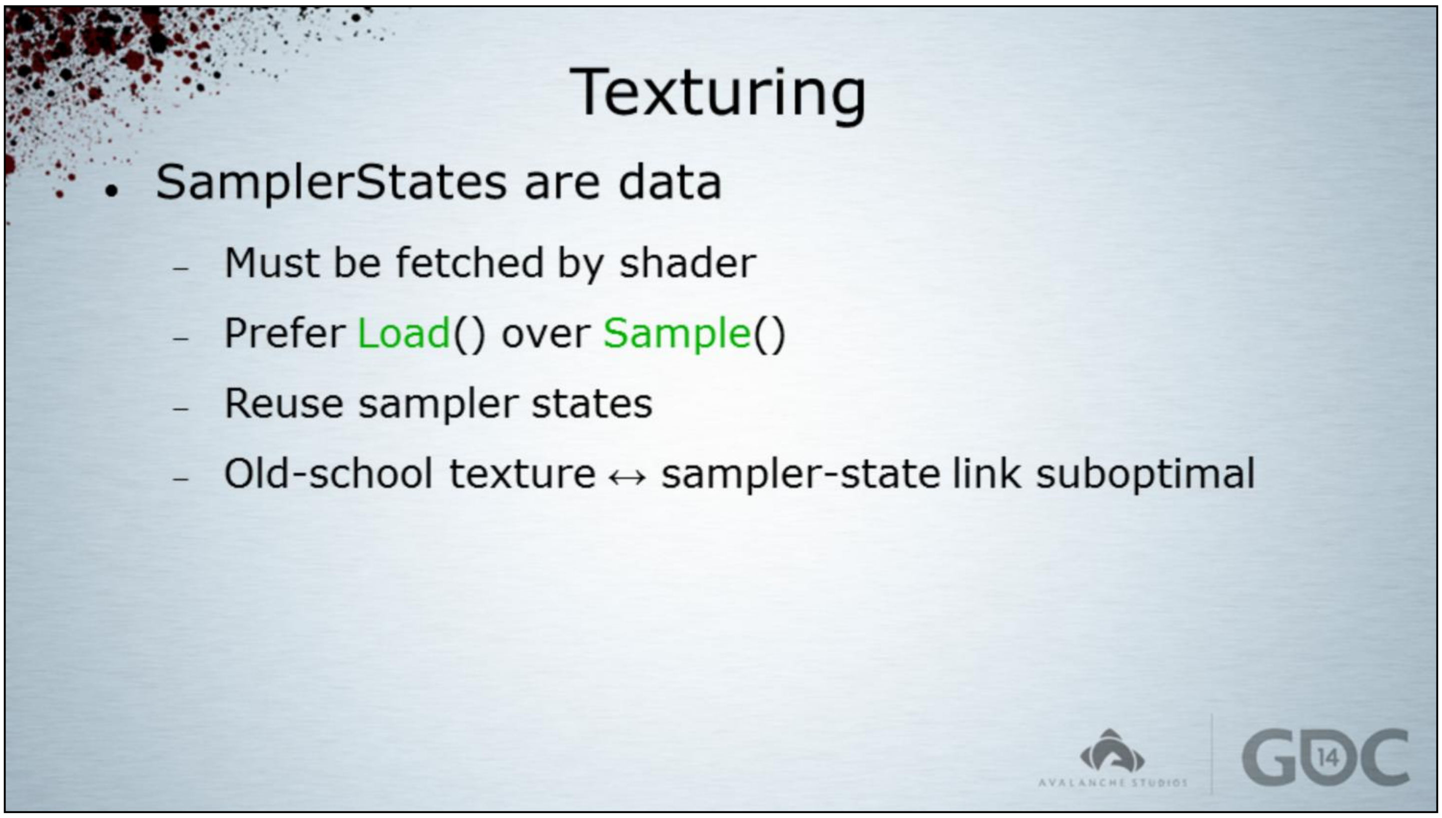


Float

- Prefer conditional assignment
  - sign() - Horribly poorly implemented
  - step() - Confusing code and suboptimal for typical case
- Special hardware features
  - min3(), max3(), med3()
    - Useful for faster reductions
    - General clamp: med3(x, min_val, max_val)

As I mentioned in my talk last year, sign() is poorly implemented under the hood and for most practical cases does too much work. Most use cases actually do not care about handling 0, or would rather flip it in either direction instead of returning 0, but sign() is a generic built-in function and does generic things. Implementing it as a conditional assignment is much faster, like (x >= 0)? 1 : -1. Similarly the step() function can be implemented as (x >= a)? 1 : 0. The advantage of doing this explicitly, other than improved readability (IMHO), is that the typical use case can also be further optimized. Normally sign() and step() are multiplied with something afterward rather than just used as is. If you use the functions, not only are you paying for the suboptimal handling under the hood, you are also paying for that multiplication. However, when written as a conditional assignment you can simply merge that into your parameters that you select from and get it for free.

GCN has 3-component min, max and med instructions. The obvious usage case is to do faster reductions, e.g. finding the max value in a 3x3 neighborhood would take 4 cycles instead of 8. A somewhat less obvious use case is to use med3() as a general clamp in one cycle instead of two.

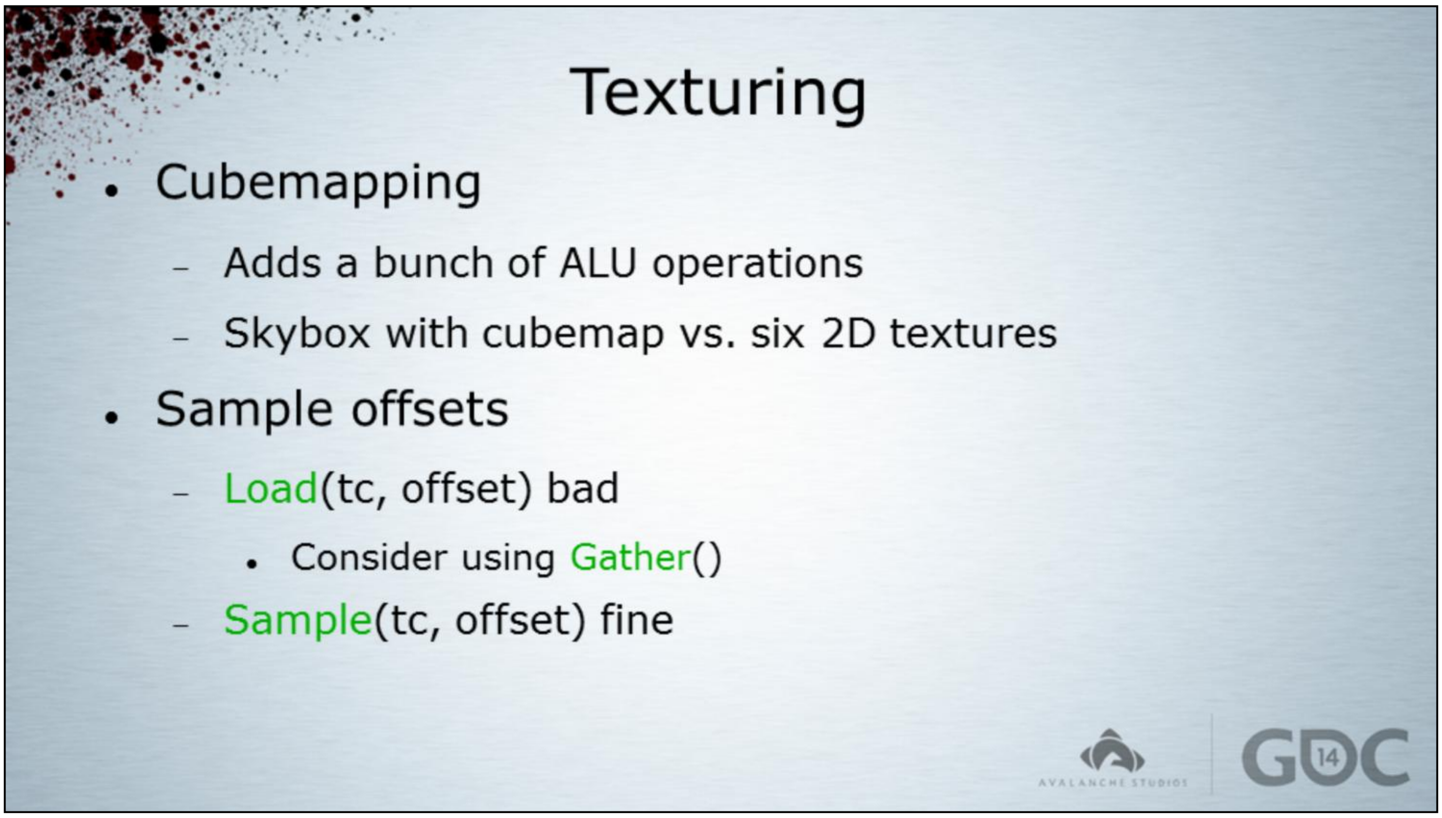


Texturing

- SamplerStates are data
  - Must be fetched by shader
  - Prefer Load() over Sample()
  - Reuse sampler states
  - Old-school texture ↔ sampler-state link suboptimal

SamplerStates are data that must be fetched. Thus it is preferable to use Load() instead of Sample() with a point filter if that works and without introducing conversions between floats and ints (which may be more costly as they are vector instructions).

If you are coming from a DX9 or last-gen console engine, chances are that sampler states and textures are linked, i.e. texture 0 uses sampler state 0 and texture 1 uses sampler state 1 etc.. Already DX10 stepped away from this model, but it is easy to get stuck with the same conceptual model as long as your engine has last-gen console support or uses shaders with DX9-style syntax. Typically a shader has many more textures than sampler states, so the result is that the shader will have to load many more sampler states than is necessary and will simply keep many copies of the same data in its registers, wasting scalar register space and loads.

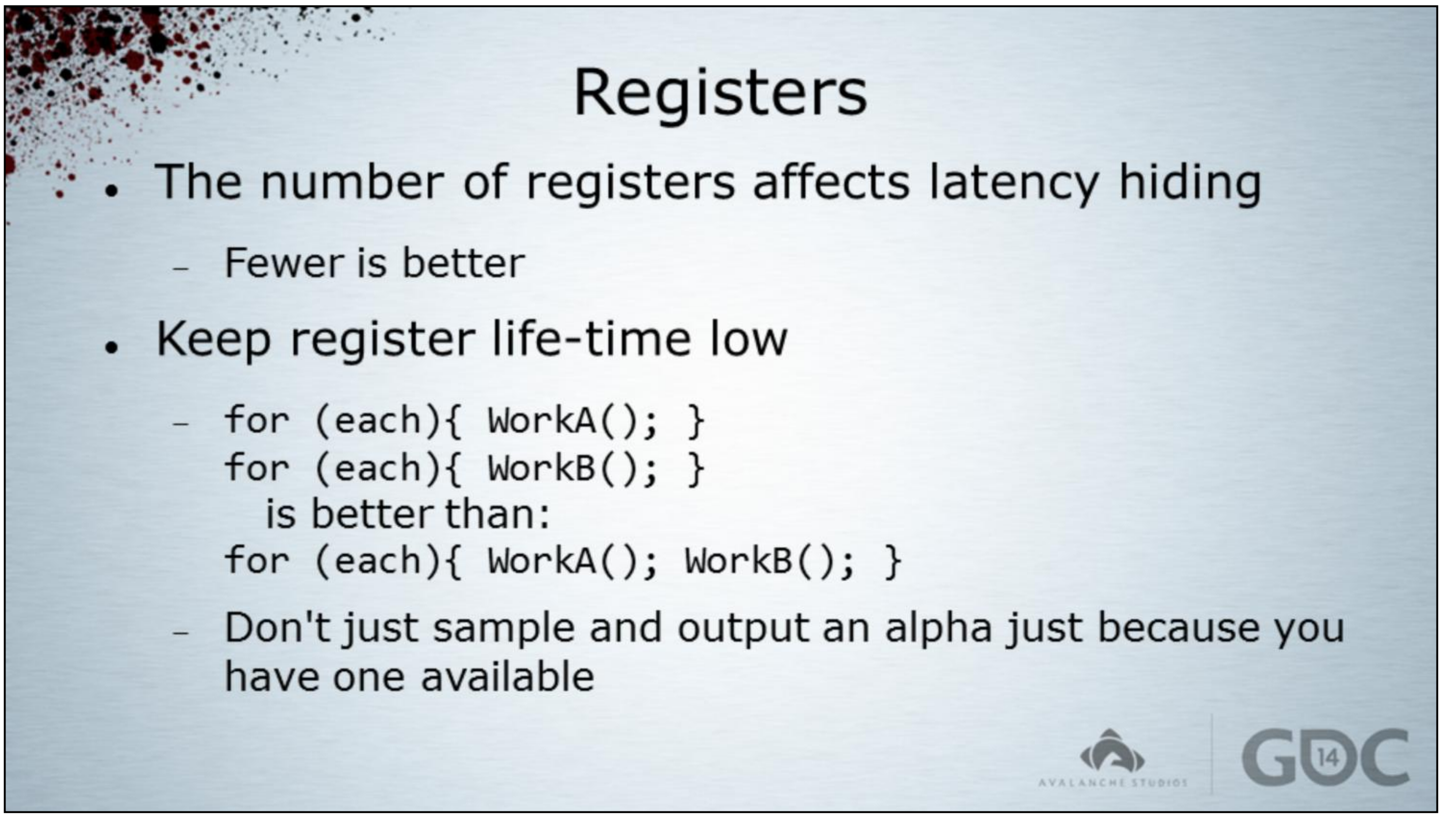


Texturing

- Cubemapping
  - Adds a bunch of ALU operations
  - Skybox with cubemap vs. six 2D textures
- Sample offsets
  - Load(tc, offset) bad
    - Consider using Gather()
  - Sample(tc, offset) fine

As mentioned before, cubemap sampling comes at a bit of overhead in terms of ALU instructions. This can typically not be avoided since you usually really need a cubemap where you use them. The exception is things like a skybox. While storing it as a cubemaps seems reasonable, it can be faster to simply draw it using a set of 2D textures or an atlas.

Load() and Sample() both accept integer offsets for sampling neighboring texels. In previous generations this was baked into the instruction and came for free. For GCN this is no longer true for Load(). As a result, sampling with an offset inserts ALU instructions to add the offsets between each Load(). In many cases like this, it makes sense to use Gather() instead. For Sample() though, providing an offset is still OK and will still be for free.

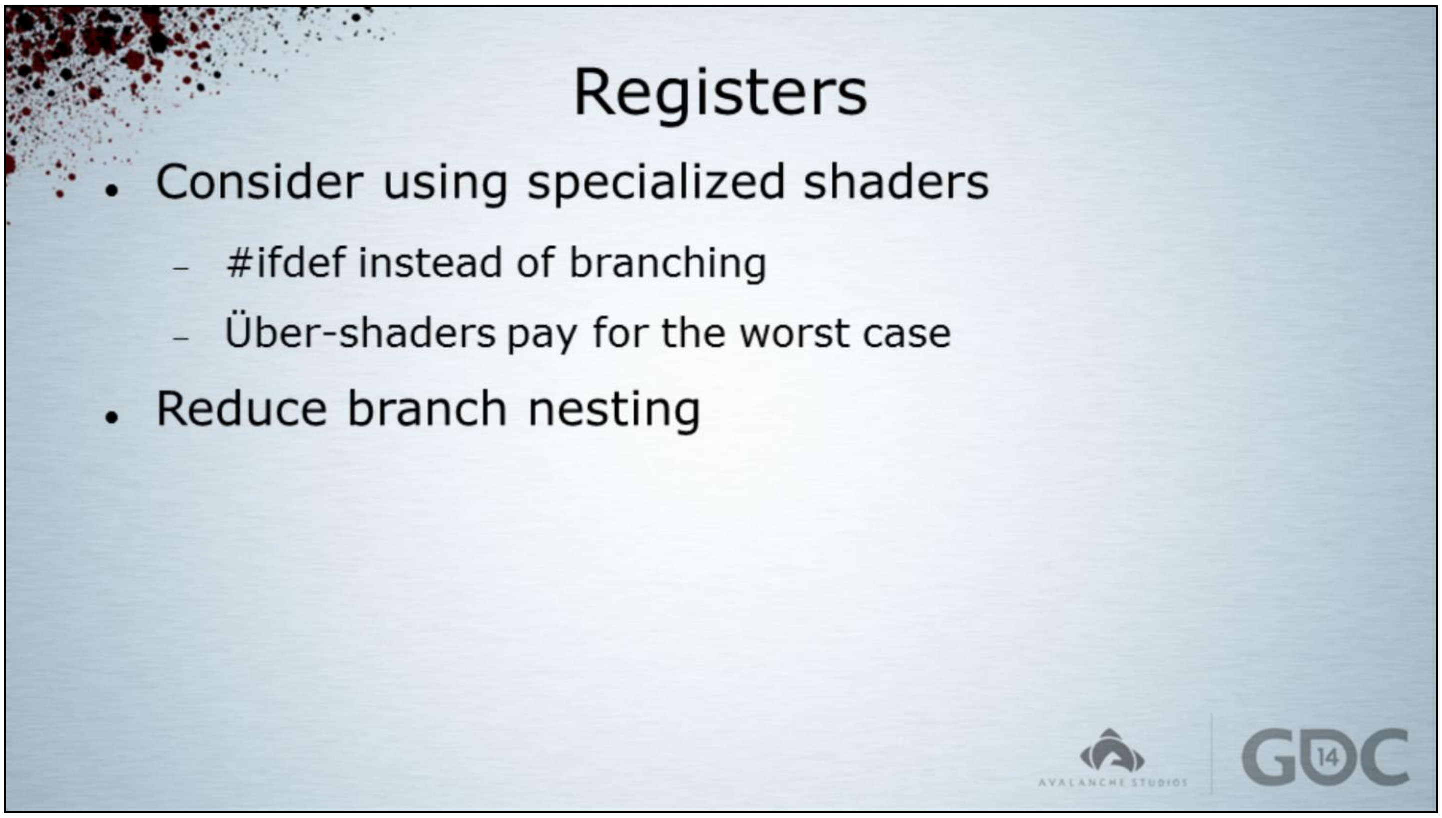


Registers

- The number of registers affects latency hiding
  - Fewer is better
- Keep register life-time low

  -
    ```
    for (each){ WorkA(); }
    for (each){ WorkB(); }
    ```
    is better than:
    ```
    for (each){ WorkA(); WorkB(); }
    ```
  - Don't just sample and output an alpha just because you have one available

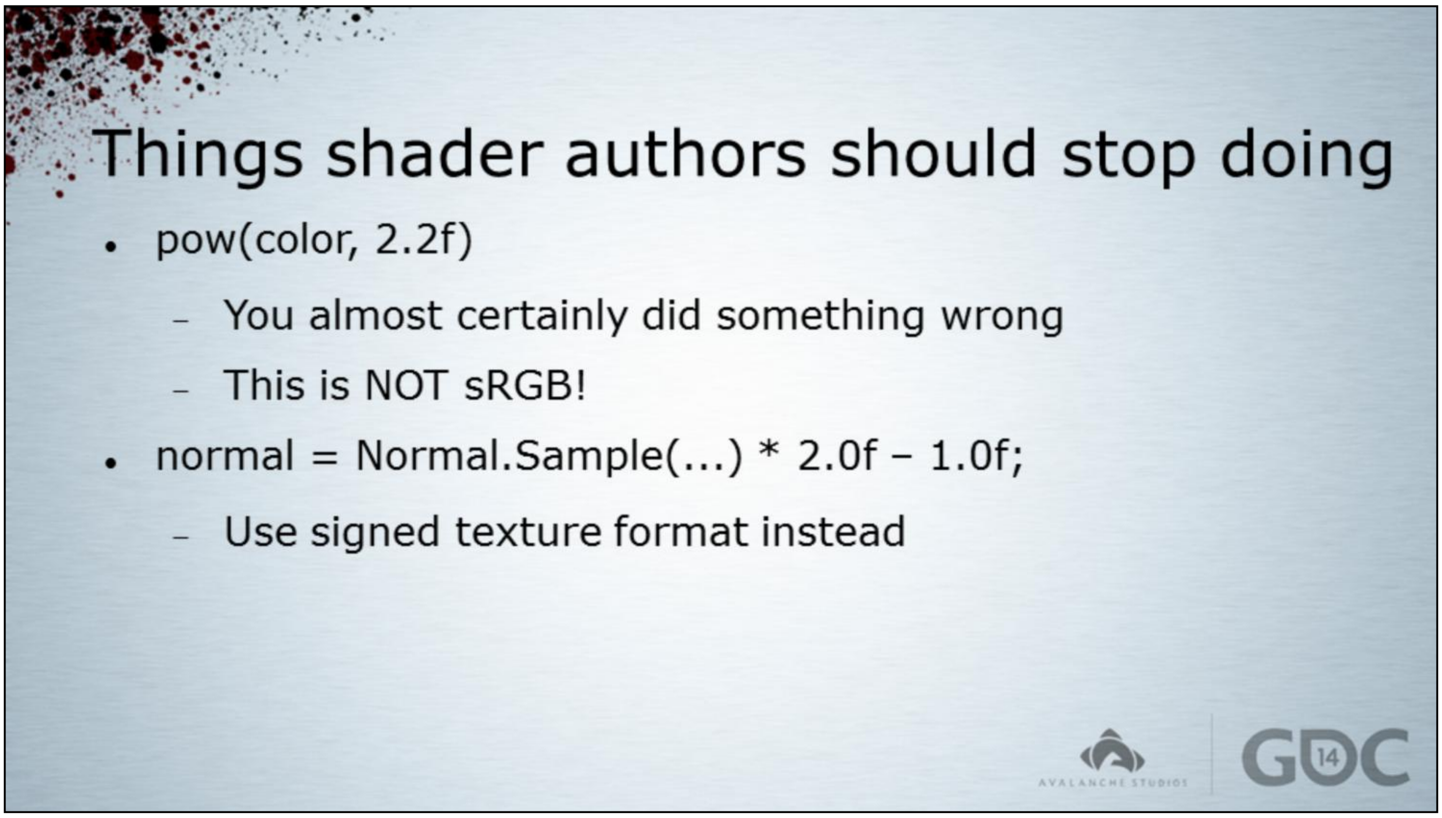
Optimizing for registers is unfortunately one of the hardest things to give solid advice on. This is mostly a black box completely under the control of the compiler and not something that is particularly deterministic. ALU expressions tend to give quite predictable results, but the number of registers can go up and down in non-intuitive ways depending on the code you write. However, there are a few piece of advice one can give. The idea is overall to keep the shader "thin" in terms of the data it needs around and the stuff that it does. You will get as many registers as your worst-case path through the shader. So if you need to do two sets of work, it may be better to do one thing first and then the other, rather than both together and risking a fat path when data for both types of work need to be simultaneously resident. Keep the life-time of data as low as possible. Fetch data near its use. Keep data packed until you need it. And do not just keep data around if you have no need for it. The typical case would be to hold on to an alpha value you got from sampling a texture and simply return it in the alpha channel by the end of the shader, even if you do not really need an alpha. This will occupy a register for the entire span from sampling the texture, until the end of the shader.



Registers

- Consider using specialized shaders
  - #ifdef instead of branching
  - Über-shaders pay for the worst case
- Reduce branch nesting

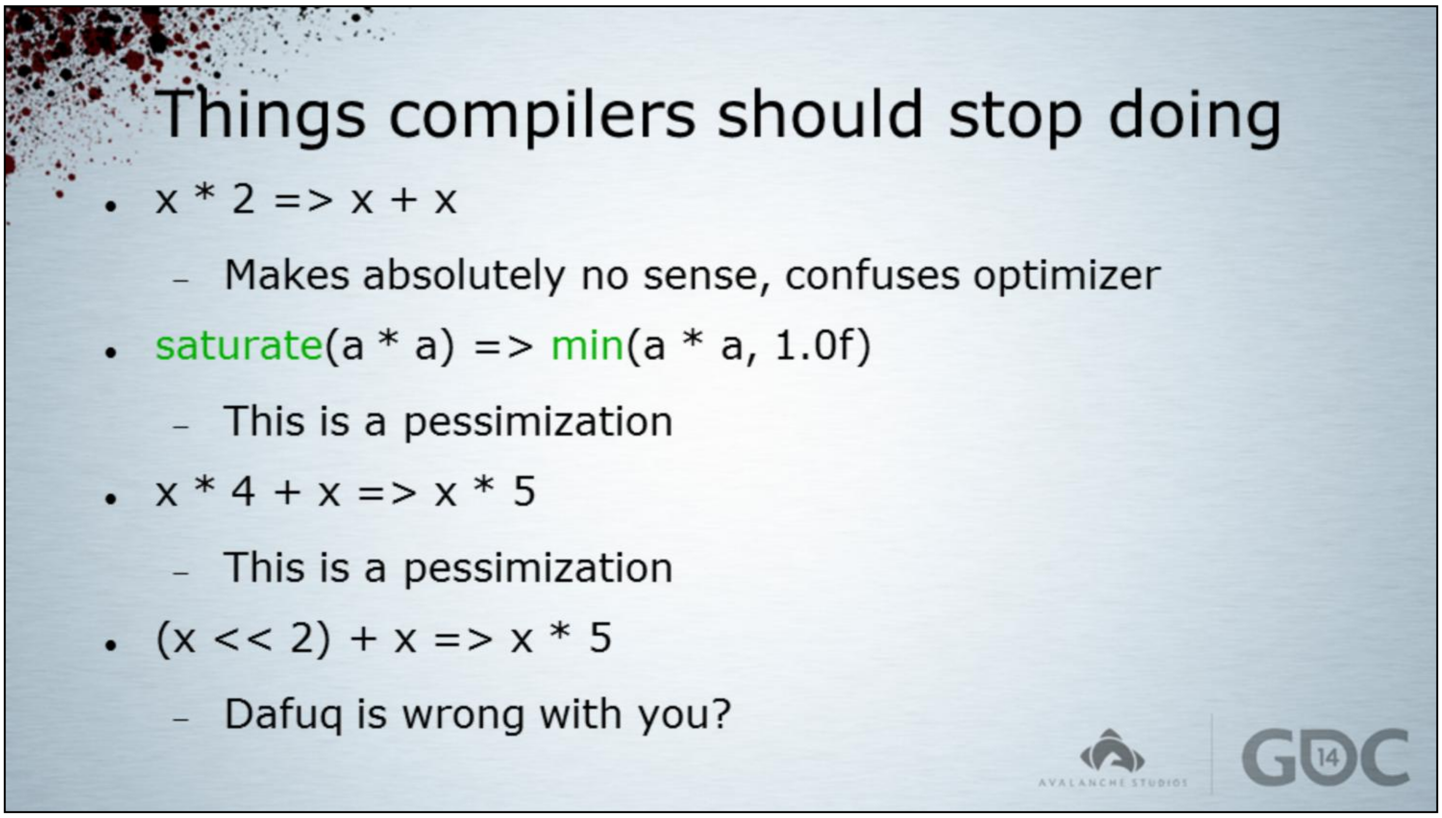
Über-shaders are great for authoring; however, not so much for execution. Branching unfortunately tends to increase register pressure. So while branches are cheap on GCN, and branching for feature selection would be a viable approach from that point of view, it can have a very negative impact on register count, and thus performance. You will always pay for your worst-case path, so even if you end up skipping the vast majority of shader work you may still get a significant slowdown from reduced latency hiding. So it is advisable to create a set of specialized shaders, perhaps with #ifdef in the Über-shader.



Things shader authors should stop doing

- pow(color, 2.2f)
  - You almost certainly did something wrong
  - This is NOT sRGB!
- normal = Normal.Sample(...) * 2.0f – 1.0f;
  - Use signed texture format instead

The rant part of this talk!



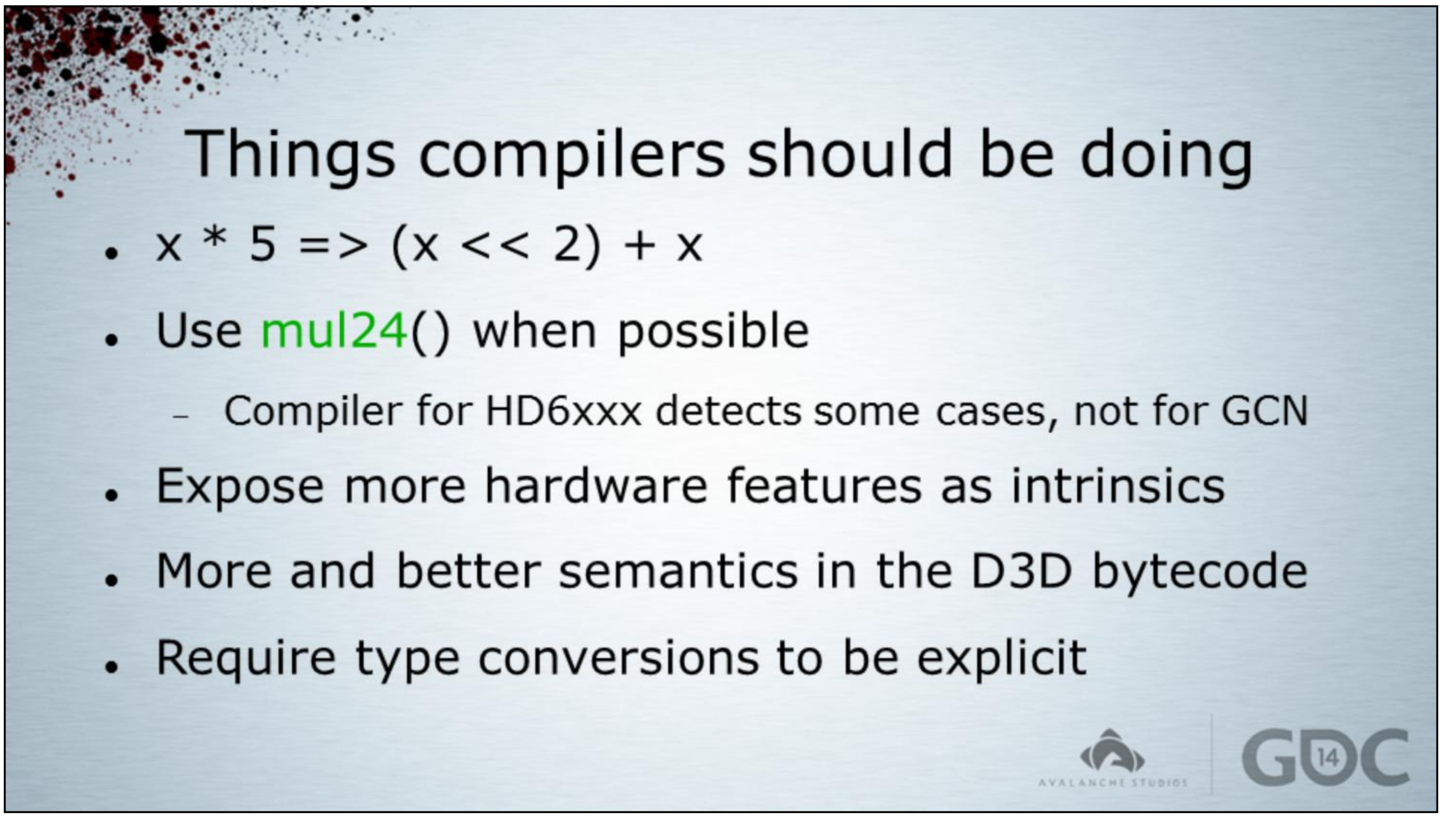
Things compilers should stop doing

- x * 2 => x + x
  - Makes absolutely no sense, confuses optimizer
- saturate(a * a) => min(a * a, 1.0f)
  - This is a pessimization
- x * 4 + x => x * 5
  - This is a pessimization
- (x << 2) + x => x * 5
  - Dafuq is wrong with you?

Compilers are sometimes too smart for their own good. In the case of fxc for windows, the biggest issue is that it has no knowledge about the hardware, it only works with an abstract pseudo-instruction set. From that point of view merging things to multiply integer by 5 instead of the intended shift and add makes sense, except on all hardware a 32bit multiply is much slower than shifts and adds.

Things compilers should stop doing

- asfloat(0x7FFFFF) => 0
  - This is a bug. It's a cast. Even if it was a MOV it should still preserve all bits and not flush denorms.
- Spend awful lots of time trying to unroll loops with [loop] tag
  - I don't even understand this one
- Treat vectors as anything else than a collection of floats



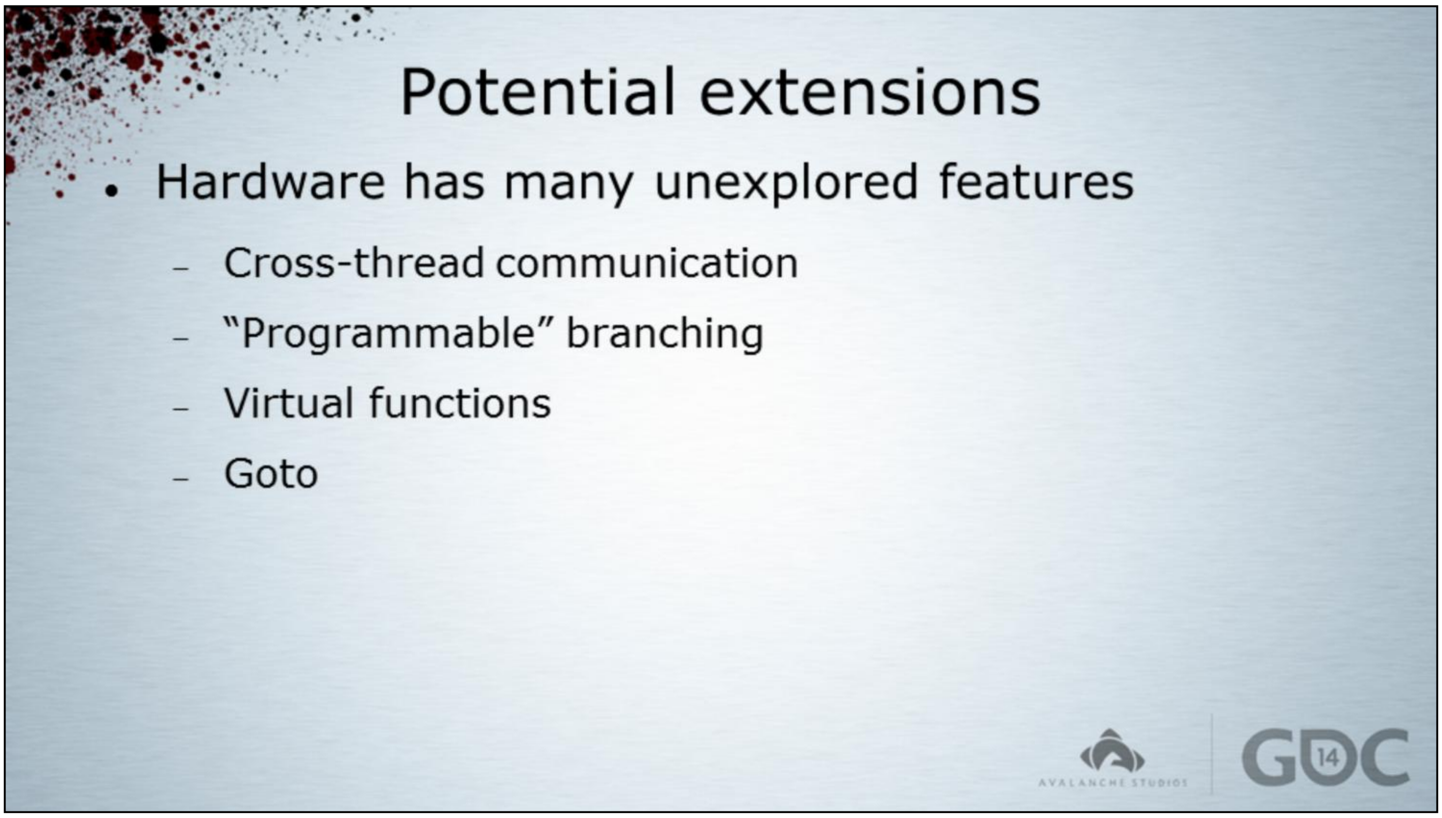
Things compilers should be doing

- x * 5 => (x << 2) + x
- Use mul24() when possible
  - Compiler for HD6xxx detects some cases, not for GCN
- Expose more hardware features as intrinsics
- More and better semantics in the D3D bytecode
- Require type conversions to be explicit

Obviously, what the compiler should be doing is the reverse, take a multiply by 5 and convert to a shift and add. This will be an improvement that works for a limited number of integers, but for many common small numbers it works, as well as some large ones.

For PC the D3D bytecode needs to provide more semantics to allow the driver optimizer to do a better job. Information about input ranges and so on that fxc knows about is lost in the process.

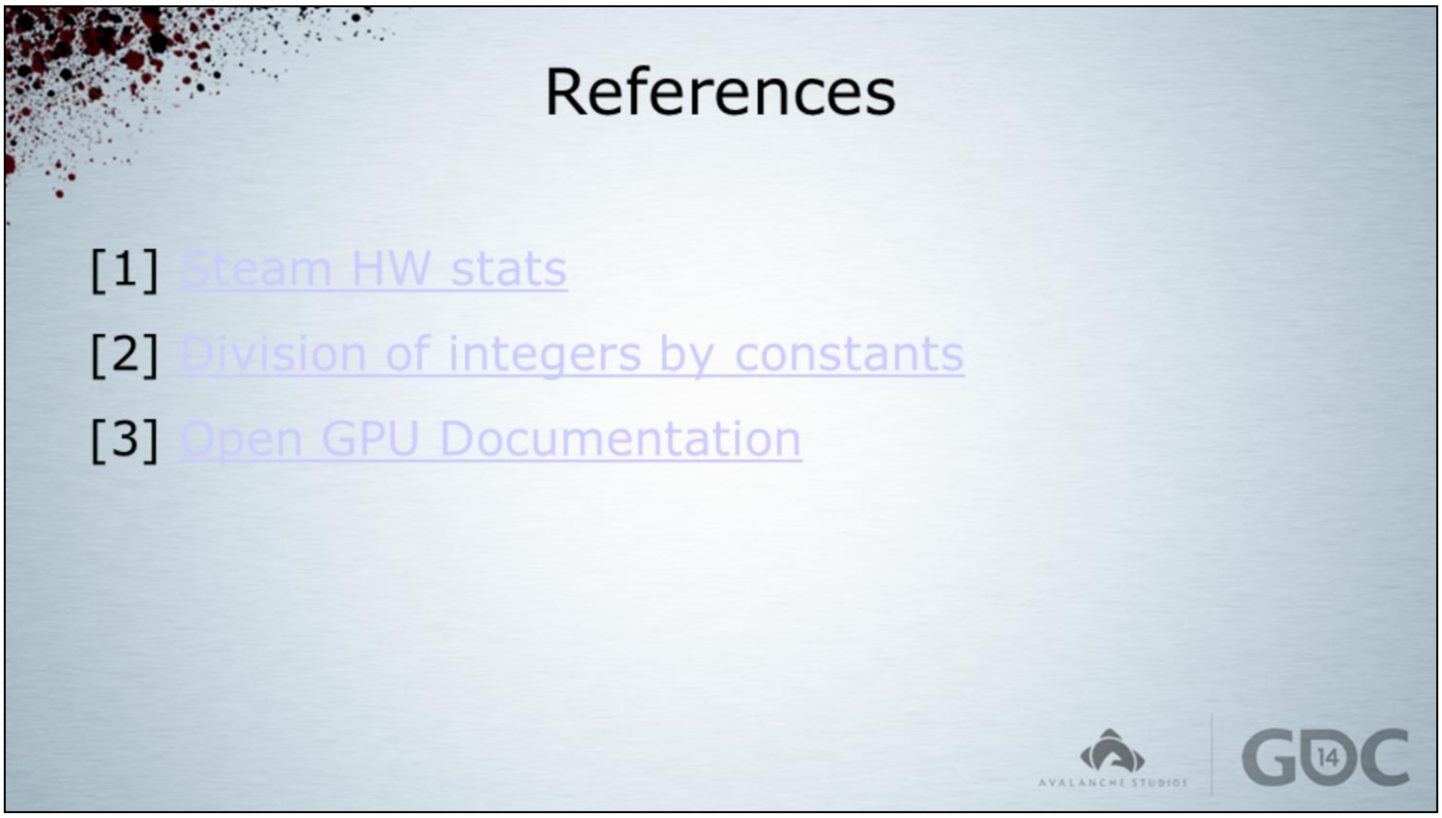
As a general complaint on HLSL, the language is way to permissive on accepting implicit type conversions. Make a function that takes a float4x4 matrix, call it with a bool and it just compiles and obviously does nothing like what you intended. Can we please at least require a cast?



Potential extensions

- Hardware has many unexplored features
  - Cross-thread communication
  - "Programmable" branching
  - Virtual functions
  - Goto

There are lots of things on the hardware that would be interesting to explore in the future. Some of these things are just kind of cool in a sense, but without a clear use case. Such as passing data between lanes in a shader. What would you use it for, other than gradients? I don't know. The data for branching just lies in a scalar register. What can we use it for? Debug visualization of branch coherency perhaps? I am sure we will discover many interesting opportunities in the future.



References

[1] Steam HW stats

[2] Division of integers by constants

[3] Open GPU Documentation

The GCN architecture is surprisingly well documented. If you enjoyed this talk, I would recommend you take a look at the documentation. A lot of the content was inspired from that.

Questions?

Twitter: _Humus_

Email: emil.persson@avalanchestudios.se