Low-level Thinking in High-level Shading Languages

Emil Persson
Head of Research, Avalanche Studios

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29
2013

Problem formulation

“Nowadays renowned industry luminaries include shader snippets in their GDC presentations where trivial transforms would have resulted in a faster shader”

This topic has grown on me over the years as I have seen shader code on slides at conferences, by brilliant people, where the code could have been written in a much better way. Occasionally I hear an “this is unoptimized” or “educational example” attached to it, but most of the time this excuse doesn't hold. I sometimes sense that the author may use “unoptimized” or “educational” as an excuse because they are unsure how to make it right. And then again, code that's shipping in SDK samples from IHVs aren't always doing it right either.

When the best of the best aren't doing it right, then we have a problem as an industry.

Goal of this presentation

“Show that low-level thinking is still relevant today”

Background

- In the good ol' days, when grandpa was young …
  - Shaders were short
    - SM1: Max 8 instructions, SM2: Max 64 instructions
  - Shaders were written in assembly
    - Already getting phased out in SM2 days
  - D3D opcodes mapped well to real HW
  - Hand-optimizing shaders was a natural thing to do

```
def     c0, 0.3f, 2.5f, 0, 0

texld   r0, t0
sub     r0, r0, c0.x
mul     r0, r0, c0.y
```

```
def     c0, -0.75f, 2.5f, 0, 0

texld   r0, t0
mad     r0, r0, c0.y, c0.x
```

$$(x - 0.3) * 2.5 = x * 2.5 + (-0.75)$$

Background

- Low-level shading languages are dead
  - Unproductive way of writing shaders
  - No assembly option in DX10+
    - Nobody used it anyway
  - Compilers and driver optimizers do a great job (sometimes ...)
  - Hell, these days artists author shaders!
    - Using visual shader editors
      - With boxes and arrows
        - Without counting cycles, or inspecting the asm
        - Without even consulting technical documentation
        - Argh, the kids these day! Back in my days we had...
- Consequently:
  - Shader writers have lost touch with the HW

Assembly languages are dead. The last time I used one was 2003. Since then it has been HLSL and GLSL for everything. I haven't looked back.

So shading has of course evolved, and it is a natural development that we are seeing higher level abstractions as we're moving along. Nothing wrong with that. But as the gap between the hardware and the abstractions we are working with widens, there is an increasing risk of losing touch with the hardware. If we only ever see the HLSL code, but never see what the GPU runs, this will become a problem. The message in this presentation is that maintaining a low-level mindset while working in a high-level shading language is crucial for writing high performance shaders.

Why bother?

- How your shader is written matters!

```
//      float3    float      float    float3       float  float
return Diffuse * n_dot_l * atten * LightColor * shadow * ao;
```

```
0  x: MUL_e       ____,  R0.z,  R0.w
   y: MUL_e       ____,  R0.y,  R0.w
   z: MUL_e       ____,  R0.x,  R0.w
1  y: MUL_e       ____,  R1.w,  PV0.x
   z: MUL_e       ____,  R1.w,  PV0.y
   w: MUL_e       ____,  R1.w,  PV0.z
2  x: MUL_e       ____,  R1.x,  PV1.w
   z: MUL_e       ____,  R1.z,  PV1.y
   w: MUL_e       ____,  R1.y,  PV1.z
3  x: MUL_e       ____,  R2.x,  PV2.w
   y: MUL_e       ____,  R2.x,  PV2.x
   w: MUL_e       ____,  R2.x,  PV2.z
4  x: MUL_e      R2.x,  R2.y,  PV3.y
   y: MUL_e      R2.y,  R2.y,  PV3.x
   z: MUL_e      R2.z,  R2.y,  PV3.w
```

```
//      float     float     float  float    float3    float3
return (n_dot_l * atten) * (shadow * ao) * (Diffuse * LightColor);
```

```
0  x: MUL_e       ____,  R2.x,  R2.y
   y: MUL_e      R0.y,  R0.y,  R1.y      VEC_021
   z: MUL_e      R0.z,  R0.x,  R1.x      VEC_120
   w: MUL_e       ____,  R0.w,  R1.w
   t: MUL_e      R0.x,  R0.z,  R1.z
1  w: MUL_e       ____,  PV0.x,  PV0.w
2  x: MUL_e      R0.x,  R0.z,  PV1.w
   y: MUL_e      R0.y,  R0.y,  PV1.w
   z: MUL_e      R0.z,  R0.x,  PV1.w
```

This is a clear illustration of why we should bother with low-level thinking. With no other change than moving things around a little and adding some parentheses we achieved a substantially faster shader. This is enabled by having an understanding of the underlying HW and mapping of HLSL constructs to it.

The HW used in this presentation is a Radeon HD 4870 (selected because it features the most readable disassembly), but most of everything in this slide deck is really general and applies to any GPU unless stated otherwise.

Why bother?

- Better performance
  - “We're not ALU bound ...”
    - Save power
    - More punch once you optimize for TEX/BW/etc.
    - More headroom for new features
  - “We'll optimize at the end of the project …”
    - Pray that content doesn't lock you in ...
- Consistency
  - There is often a best way to do things
  - Improve readability
- It's fun!

Hardware comes in many configurations that are balanced differently between sub-units. Even if you are not observing any performance increase on your particular GPU, chances are there is another configuration on the market where it makes a difference.

Reducing utilization of ALU from say 50% to 25% while bound by something else (TEX/BW/etc.) probably doesn't improve performance, but lets the GPU run cooler. Alternatively, with today's fancy power-budget based clocks could let the hardware maintain a higher clock-rate than it could otherwise, and thereby still run faster.

”The compiler will optimize it!”

”The compiler will optimize it!”

- Compilers are cunning!
  - Smart enough to fool themselves!
- However:
  - They can't read your mind
  - They don't have the whole picture
  - They work with limited data
  - They can't break rules
    - Well, mostly … (they can make up their own rules)

Compilers only understand the semantics of the operations in the shader. They don't know what you are trying to accomplish. Many possible optimizations are “unsafe” and must thus be done by the shader author.

”The compiler will optimize it!”

Will it go mad? (pun intended)

```
float main(float x : TEXCOORD) : SV_Target
{
    return (x + 1.0f) * 0.5f;
}
```

This is the most trivial example of an piece of code you may think could be optimized automatically to use a MAD instruction instead of ADD + MUL, because both constants are compile time literals and overall very friendly numbers.

”The compiler will optimize it!”

Will it go mad? (pun intended)

```
float main(float x : TEXCOORD) : SV_Target
{
    return (x + 1.0f) * 0.5f;
}
```

Nope!

```
add r0.x, v0.x, l(1.000000)
mul o0.x, r0.x, l(0.500000)
```

What about the driver?

Turns out fxc is still not comfortable optimizing it.

GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

"The compiler will optimize it!"

Will it go mad? (pun intended)

```
float main(float x : TEXCOORD) : SV_Target
{
    return (x + 1.0f) * 0.5f;
}
```

Nope!

```
add r0.x, v0.x, l(1.000000)
mul o0.x, r0.x, l(0.500000)
```

Nope!

```
00 ALU: ADDR(32) CNT(2)
      0  y: ADD          ____,  R0.x,  1.0f
      1  x: MUL_e        R0.x,  PV0.y,  0.5
01 EXP_DONE: PIX0, R0.x___
```

The driver is bound by the semantics of the provided D3D byte-code. Final code for the GPU is exactly what was written in the shader.

You will see the same results on PS3 too, except in this particular case it seems comfortable turning it into a MAD. Probably because the constant 1.0f there. Any other constant and it behaves just like PC here.

The Xbox360 shader compiler is a funny story. It just doesn't care. It does this optimization anyway, always, even when it obviously breaks stuff. It will slap things together even if the resulting constant overflows to infinity, or underflows to become zero. 1.#INF is your constant and off we go! Oh, zero, I only need to do a MUL then, yay! There are of course many more subtle breakages because of this, where you simply lost a whole lot of floating point precision due to the change and it's not obvious why.

Why not?

- The result might not be exactly the same
- May introduce INFs or NaNs
- Generally, the compiler is great at:
  - Removing dead code
  - Eliminating unused resources
  - Folding constants
  - Register assignment
  - Code scheduling
- But generally does not:
  - Change the meaning of the code
  - Break dependencies
  - Break rules

We are dealing with IEEE floats here. Changing the order of operations is NOT safe. In the best case we get the same result. We might even gain precision if order is changed. But it could also get worse, depending on the values in question. Worst case it breaks completely because of overflow or underflow, or you might even get a NaN where the unoptimized code works.

Consider x = 0.2f in this case:

sqrt(0.1f * (0.2f - x)) returns exactly zero

sqrt(0.02f - 0.1f * x) returns NaN

The reason this breaks is because the expression in the second case returns a slightly negative value under the square-root. Keep in mind that neither of 0.1f, 0.2f or 0.02f can be represented exactly as an IEEE float. The deviation comes from having properly rounded constants. It's impossible for the compiler to predict these kinds of failures with unknown inputs.

Therefore:

Write the shader the way you want the hardware to run it!

That means:

Low-level thinking

Relying on the shader compiler to fix things up for you is just naïve. It generally doesn't work that way. What you write is what you get. That's the main principle to live by.

Rules

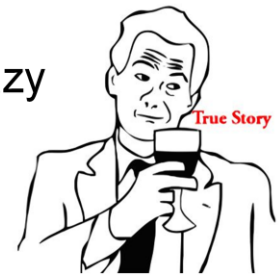
- D3D10+ generally follows IEEE-754-2008 [1]
- Exceptions include [2]:
  - 1 ULP instead of 0.5
  - Denorms flushed on math ops
    - Except MOVs
    - Min/max flush on input, but not necessarily on output
- HLSL compiler ignores:
  - The possibility of NaNs or INFs
    - e.g. x * 0 = 0, despite NaN * 0 = NaN
    - Except with precise keyword or IEEE strictness enabled
    - Beware: compiler may optimize away your isnan() and isfinite() calls!

While the D3D compiler allows itself to ignore the possibility of INF and NaN at compile time (which is desirable in general for game development), that doesn't mean the driver is allowed to do so at runtime. If the D3D byte-code says “multiply by zero”, that's exactly what the GPU will end up doing.

Universal* facts about HW

- Multiply-add is one instruction – Add-multiply is two
- abs, negate and saturate are free
  - Except when their use forces a MOV
- Scalar ops use fewer resources than vector

- Shader math involving only constants is crazy
- Not doing stuff is faster than doing stuff



* For a limited set of known universes

This has been true on all GPUs I have ever worked with. Doesn't mean there couldn't possibly be an exception out there, but I have yet to see one.

Some early ATI cards had a pre-adder such that add-multiply could be a single instruction in specific cases. There were some restrictions though, like no swizzles and possibly others. It was intended for fast lerps IIRC. But even so, if you did multiply-add instead of add-multiply you freed up the pre-adder for other stuff, so the recommendation still holds.

MAD

- Any linear ramp → mad
  - With a clamp → mad_sat
    - If clamp is not to [0, 1] → mad_sat + mad
  - Remapping a range == linear ramp
- MAD not always the most intuitive form
  - MAD = x * slope + offset_at_zero
  - Generate slope & offset from intuitive params

(x – start) * slope	→	x * slope + (-start * slope)
(x – start) / (end – start)	→	x * (1.0f / (end - start)) + (-start / (end - start))
(x – mid_point) / range + 0.5f	→	x * (1.0f / range) + (0.5f - mid_point / range)
clamp($s_1 + (x-s_0)*(e_1-s_1)/(e_0-s_0)$, s_1, e_1)	→	saturate(x * (1.0f/(e_0-s_0)) + (-s_0/(e_0-s_0))) * (e_1-s_1) + s_1

Any sort of remapping of one range to another should normally be a single MAD instruction, possibly with a clamp, or in the most general case be MAD_SAT + MAD.

The examples here are color-coded to show what the slope and offset parts are. Left is the “intuitive” notation, and right is the optimized.

Example 1: Starting point and slope from there.

Example 2: Mapping start to end into 0-1 range

Example 3: Mapping a range around midpoint to 0-1

Example 4: Fully general remapping of [s0, e0] range to [s1, e1] range with clamping.

MAD

- More transforms

```
x * (1.0f – x)          →   x – x * x
x * (y + 1.0f)          →   x * y + x
(x + c) * (x - c)       →   x * x + (-c * c)
(x + a) / b             →   x * (1.0f / b) + (a / b)
x += a * b + c * d;     →   x += a * b;
                            x += c * d;
```

More remapping of expressions. All just standard math, nothing special here.

The last example may surprise you, but that's 3 instructions as written on the left (MUL-MAD-ADD), and 2 on the right (MAD-MAD). This is because the semantics of the expression dictates that (a*b+c*d) is evaluated before the += operator.

Division

- a / b typically implemented as a * rcp(b)
  - D3D asm may use DIV instruction though
  - Explicit rcp() sometimes generates better code
- Transforms

a / (x + b)	→	rcp(x * (1.0f / a) + (b / a))
a / (x * b)	→	rcp(x) * (a / b) rcp(x * (b / a))
a / (x * b + c)	→	rcp(x * (b / a) + (c / a))
(x + a) / x	→	1.0f + a * rcp(x)
(x * a + b) / x	→	a + b * rcp(x)

- It's all junior high-school math!
- It's all about finishing your derivations! [3]

Given that most hardware implement division as the reciprocal of the denominator multiplied with the numerator, expressions with division should be rewritten to take advantage of MAD to get a free addition with that multiply. Sadly, this opportunity is more often overlooked than not.

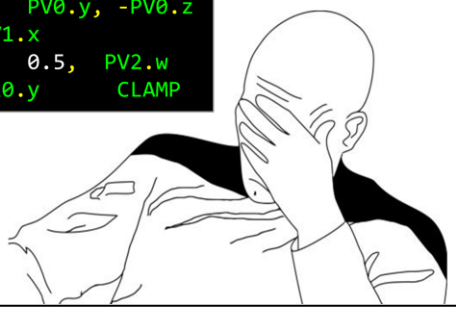
MADness

- From our code-base:

```
float AlphaThreshold(float alpha, float threshold, float blendRange)
{
    float halfBlendRange = 0.5f*blendRange;
    threshold = threshold*(1.0f + blendRange) - halfBlendRange;
    float opacity = saturate( (alpha - threshold + halfBlendRange)/blendRange );
    return opacity;
}
```

```
mul r0.x, cb0[0].y, l(0.500000)
add r0.y, cb0[0].y, l(1.000000)
mad r0.x, cb0[0].x, r0.y, -r0.x
add r0.x, -r0.x, v0.x
mad r0.x, cb0[0].y, l(0.500000), r0.x
div_sat o0.x, r0.x, cb0[0].y
```

```
0  y: ADD       ____, KC0[0].y,  1.0f
   z: MUL_e     ____, KC0[0].y,  0.5
   t: RCP_e     R0.y, KC0[0].y
1  x: MULADD_e  ____, KC0[0].x,  PV0.y, -PV0.z
2  w: ADD       ____, R0.x, -PV1.x
3  z: MULADD_e  ____, KC0[0].y,  0.5,  PV2.w
4  x: MUL_e     R0.x, PV3.z,  R0.y      CLAMP
```



A quick glance at this code may lead you to believe it's just a plain midpoint-and-range computation, like in the examples in a previous slide, but it's not. If the code would be written in MAD-form, this would be immediately apparent.

However, in the defense of this particular code, the implementation was at least properly commented with what it is actually computing. Even so, a seasoned shader writer should intuitively feel that this expression would boil down to a single MAD.

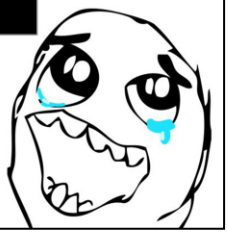
MADness

- AlphaThreshold() reimagined!

```
// scale = 1.0f / blendRange
// offset = 1.0f - (threshold/blendRange + threshold)
float AlphaThreshold(float alpha, float scale, float offset)
{
    return saturate( alpha * scale + offset );
}
```

```
mad_sat o0.x, v0.x, cb0[0].x, cb0[0].y
```

```
0  x: MULADD_e  R0.x, R0.x, KC0[0].x, KC0[0].y  CLAMP
```



As we simplify the math all the way it gets apparent that it's just a plain MAD computation. Once the scale and offset parameters are found, it's clear that they don't match the midpoint-and-range case.

Modifiers

- Free unless their use forces a MOV
  - abs/neg are on input
  - saturate is on output

```
float main(float2 a : TEXCOORD) : SV_Target
{
    return abs(a.x) * abs(a.y);
}
```

```
0  x: MUL_e       R0.x,  |R0.x|,  |R0.y|
```

```
float main(float2 a : TEXCOORD) : SV_Target
{
    return abs(a.x * a.y);
}
```

```
0  y: MUL_e       ____,  R0.x,  R0.y
1  x: MOV         R0.x,  |PV0.y|
```

You want to place abs() such that they happen on input to an operation rather than on output. If abs() is on output another operation has follow it for it to happen. If more stuff happens with the value before it gets returned, the abs() can be rolled into the next operation as an input modifier there. However, if no more operations are done on it, the compiler is forced to insert a MOV instruction.

Modifiers

- Free unless their use forces a MOV
  - abs/neg are on input
  - saturate is on output

```
float main(float2 a : TEXCOORD) : SV_Target
{
    return -a.x * a.y;
}
```

```
0  x: MUL_e       R0.x, -R0.x,  R0.y
```

```
float main(float2 a : TEXCOORD) : SV_Target
{
    return -(a.x * a.y);
}
```

```
0  y: MUL_e       ____,  R0.x,  R0.y
1  x: MOV         R0.x, -PV0.y
```

Same thing with negates.

Modifiers

- Free unless their use forces a MOV
  - abs/neg are on input
  - saturate is on output

```
float main(float a : TEXCOORD) : SV_Target
{
    return 1.0f - saturate(a);
}
```

```
0  y: MOV          ____,  R0.x      CLAMP
1  x: ADD         R0.x, -PV0.y,  1.0f
```

```
float main(float a : TEXCOORD) : SV_Target
{
    return saturate(1.0f - a);
}
```

```
0  x: ADD        R0.x, -R0.x,  1.0f   CLAMP
```

saturate() on the other hand is on output. So you should avoid calling it directly on any of your inputs (interpolators, constants, texture fetch results etc.), but instead try to roll any other math you need to do on it inside the saturate() call. This is not always possible, but prefer this whenever it works.

Modifiers

- saturate() is free, min() & max() are not
  - Use saturate(x) even when max(x, 0.0f) or min(x, 1.0f) is sufficient
    - Unless (x > 1.0f) or (x < 0.0f) respectively can happen and matters
  - Unfortunately, HLSL compiler sometimes does the reverse …
    - saturate(dot(a, a)) → "Yay! dot(a, a) is always positive" → min(dot(a, a), 1.0f)
    - Workarounds:
      - Obfuscate actual ranges from compiler
        - e.g. move literal values to constants
      - Use precise keyword
        - Enforces IEEE strictness
        - Be prepared to work around the workaround and triple-check results
        - The mad(x, slope, offset) function can reinstate lost MADs

Most of the time the HLSL compiler doesn't know the possible range of values in a variable. However, results from saturate() and frac() are known to be in [0,1], and in some cases it can know a variable is non-negative or non-positive due to the math (ignoring NaNs). It is also possible to declare unorm float (range [0, 1]) and snorm float (range [-1, 1]) variables to tell the compiler the expected range. Considering the shenanigans with saturate(), these hints may actually de-optimize in many cases.

HLSL compiler workaround

- Using precise keyword
  - Compiler can no longer ignore NaN
  - saturate(NaN) == 0

```
float main(float3 a : TEXCOORD0) : SV_Target
{
    return saturate(dot(a, a));
}
```

```
dp3 r0.x, v0.xyzx, v0.xyzx
min o0.x, r0.x, l(1.000000)
```

```
0 x: DOT4_e   ____, R0.x, R0.x
  y: DOT4_e   ____, R0.y, R0.y
  z: DOT4_e   ____, R0.z, R0.z
  w: DOT4_e   ____, (0x80000000, -0.0f).x, 0.0f
1 x: MIN_DX10 R0.x, PV0.x, 1.0f
```

```
float main(float3 a : TEXCOORD0) : SV_Target
{
    return (precise float) saturate(dot(a, a));
}
```

```
dp3_sat o0.x, v0.xyzx, v0.xyzx
```

```
0 x: DOT4_e R0.x, R0.x, R0.x      CLAMP
  y: DOT4_e ____, R0.y, R0.y      CLAMP
  z: DOT4_e ____, R0.z, R0.z      CLAMP
  w: DOT4_e ____, (0x80000000, -0.0f).x, 0.0f CLAMP
```

The reason precise works is that it enforces IEEE strictness for that expression. saturate(x) is defined as min(max(x, 0.0f), 1.0f). If x is NaN the result should be 0. This is because min or max with one parameter as NaN returns the other parameter according to the IEEE-754-2008 specification. So max(NaN, 0.0f) = 0.0f. Would this be optimized away the final result would be 1.0f instead in this case.

This is rare case of precise actually improving performance rather than reducing it. Naturally, the preferred way would be for the compiler to treat saturate() as a first-class citizen rather than as a sequence of max and min, which would have avoided this problem in the first place.

Built-in functions

- rcp(), rsqrt(), sqrt()* map directly to HW instructions
- Equivalent math may not be optimal …
  - 1.0f / x tends to yield rcp(x)
  - 1.0f / sqrt(x) yields rcp(sqrt(x)), NOT rsqrt(x)!
- exp2() and log2() maps to HW, exp() and log() do not
  - Implemented as exp2(x * 1.442695f) and log2(x * 0.693147f)
- pow(x, y) implemented as exp2(log2(x) * y)
  - Special cases for some literal values of y
  - z * pow(x, y) = exp2(log2(x) * y + log2(z))
    - Free multiply if log2(z) can be precomputed
    - e.g. specular_normalization * pow(n_dot_h, specular_power)

sqrt() maps to a single instruction on DX10+ HW. Current-gen consoles do not have it, so it will be implemented as rcp(rsqrt(x)). Note that implementing sqrt(x) as x * rsqrt(x) typically is preferable to calling sqrt(x) on these platforms, whereas on DX10+ GPUs you should prefer just calling sqrt(x).

Built-in functions

- sign()
  - Takes care of zero case
    - Don't care? Use (x >= 0)? 1 : -1
    - sign(x) * y → (x >= 0)? y : -y
- sin(), cos(), sincos() map to HW
  - Some HW require a short preamble though
- asin(), acos(), atan(), atan2(), degrees(), radians()
  - You're doing it wrong!
  - Generates dozens of instructions
- cosh(), sinh(), log10()
  - Who are you? What business do you have in the shaders?

Conditional assignment is fast on all GPUs since the dawn of time. There is rarely a good reason to use sign(), or for that matter step(). A conditional assignment is not only faster, but is often also more readable.

Trigonometric functions are OK. There are valid use cases, but working with angles is often a sign that you didn't work out the math all the way through. There could be a more elegant and faster solution using say a dot-product.

Inverse trigonometric functions are almost guaranteed a sign that you're doing it wrong. Degrees? Get out of here!

Built-in functions

- mul(v, m)
  - v.x * m[0] + v.y * m[1] + v.z * m[2] + v.w * m[3]
  - MUL – MAD – MAD – MAD
- mul(float4(v.xyz, 1), m)
  - v.x * m[0] + v.y * m[1] + v.z * m[2] + m[3]
  - MUL – MAD – MAD – ADD
- v.x * m[0] + (v.y * m[1] + (v.z * m[2] + m[3]))
  - MAD – MAD – MAD

A w value of 1.0f is a very common case. This ought to be written explicitly in the shader for the benefit of the shader compiler, rather than relying on implicit 1.0f from the vertex fetch. Unfortunately, it doesn't boil down to MAD-MAD-MAD by default. With mul() decomposed and a few parentheses it can be achieved though. You could roll it into your own mul()-like function for readability.

Built-in functions

```
float4 main(float4 v : TEXCOORD0) : SV_Position
{
    return mul(float4(v.xyz, 1.0f), m);
}
```

```
0  x: MUL_e       ____,  R1.y,  KC0[1].w
   y: MUL_e       ____,  R1.y,  KC0[1].z
   z: MUL_e       ____,  R1.y,  KC0[1].y
   w: MUL_e       ____,  R1.y,  KC0[1].x
1  x: MULADD_e    ____,  R1.x,  KC0[0].w,  PV0.x
   y: MULADD_e    ____,  R1.x,  KC0[0].z,  PV0.y
   z: MULADD_e    ____,  R1.x,  KC0[0].y,  PV0.z
   w: MULADD_e    ____,  R1.x,  KC0[0].x,  PV0.w
2  x: MULADD_e    ____,  R1.z,  KC0[2].w,  PV1.x
   y: MULADD_e    ____,  R1.z,  KC0[2].z,  PV1.y
   z: MULADD_e    ____,  R1.z,  KC0[2].y,  PV1.z
   w: MULADD_e    ____,  R1.z,  KC0[2].x,  PV1.w
3  x: ADD         R1.x,  PV2.w,  KC0[3].x
   y: ADD         R1.y,  PV2.z,  KC0[3].y
   z: ADD         R1.z,  PV2.y,  KC0[3].z
   w: ADD         R1.w,  PV2.x,  KC0[3].w
```

```
float4 main(float4 v : TEXCOORD0) : POSITION
{
    return v.x*m[0] + (v.y*m[1] + (v.z*m[2] + m[3]));
}
```

```
0  z: MULADD_e    R0.z,  R1.z,  KC0[2].y,  KC0[3].y
   w: MULADD_e    R0.w,  R1.z,  KC0[2].x,  KC0[3].x
1  x: MULADD_e    ____,  R1.z,  KC0[2].w,  KC0[3].w
   y: MULADD_e    ____,  R1.z,  KC0[2].z,  KC0[3].z
2  x: MULADD_e    ____,  R1.y,  KC0[1].w,  PV1.x
   y: MULADD_e    ____,  R1.y,  KC0[1].z,  PV1.y
   z: MULADD_e    ____,  R1.y,  KC0[1].y,  R0.z
   w: MULADD_e    ____,  R1.y,  KC0[1].x,  R0.w
3  x: MULADD_e    R1.x,  R1.x,  KC0[0].x,  PV2.w
   y: MULADD_e    R1.y,  R1.x,  KC0[0].y,  PV2.z
   z: MULADD_e    R1.z,  R1.x,  KC0[0].z,  PV2.y
   w: MULADD_e    R1.w,  R1.x,  KC0[0].w,  PV2.x
```

Note that the number of instruction slots did not decrease due to a read-port limitation on constants on the HW. However, we freed up lanes that can be used for other work. In realistic cases the shader will end up using fewer instruction slots and run faster as those freed up lanes will be filled with other work.

Matrix math

- Matrices can gobble up any linear transform
  - On both ends!

```
float4 pos =
{
    tex_coord.x * 2.0f - 1.0f,
    1.0f - 2.0f * tex_coord.y,
    depth, 1.0f
};

float4 w_pos = mul(cs, mat);

float3 world_pos = w_pos.xyz / w_pos.w;
float3 light_vec = world_pos - LightPos;
```

```
// tex_coord pre-transforms merged into matrix
float4 pos = { tex_coord.xy, depth, 1.0f };

float4 l_pos = mul(pos, new_mat);

// LightPos translation merged into matrix
float3 light_vec = l_pos.xyz / l_pos.w;
```

```
// CPU-side code
float4x4 pre_mat = Scale(2, -2, 1) * Translate(-1, 1, 0);
float4x4 post_mat = Translate(-LightPos);

float4x4 new_mat = pre_mat * mat * post_mat;
```

Here we are converting a screen-space texture coordinate and depth value into a world-space coordinate, which is then used for computing a light vector. These transforms can be merged into the same matrix. Naturally chained matrix transforms can also be merged into the same matrix. We have had real shaders where merging the transforms ended up more than doubling the performance.

Scalar math

- Modern HW have scalar ALUs
  - Scalar math always faster than vector math
- Older VLIW and vector ALU architectures also benefit
  - Often still makes shader shorter
  - Otherwise, frees up lanes for other stuff
- Scalar to vector expansion frequently undetected
  - Depends on expression evaluation order and parentheses
  - Sometimes hidden due to functions or abstractions
  - Sometimes hidden inside functions

All NVIDIA DX10+ GPUs are scalar based.

AMDs GCN architecture (HD 7000 series) is scalar based. Earlier AMD DX10 and DX11 GPUs are VLIW.

Both AMD and NVIDIA DX9-level GPUs are vector based. This includes PS3 and Xbox360.

Mixed scalar/vector math

- Work out math on a low-level
  - Separate vector and scalar parts
  - Look for common sub-expressions
    - Compiler may not always be able to reuse them!
    - Compiler often not able to extract scalars from them!
    - dot(), normalize(), reflect(), length(), distance()
- Manage scalar and vector math separately
  - Watch out for evaluation order
    - Expression are evaluated left-to-right
    - Use parenthesis

normalize(), length(), distance() etc. all contain a dot() call. The compiler only generates one call if they match in code mixing these functions, but only for exact matches. For instance, if you have length(a – b) in your code, distance(a, b) will reuse the shared sub-expression, whereas for distance(b, a) it won't.

Hidden scalar math

- normalize(vec)
  - vector in, vector out, but *intermediate* scalar values
  - normalize(vec) = vec * rsqrt(dot(vec, vec))
    - dot() returns scalar, rsqrt() still scalar
    - Handle original vector and normalizing factor separately
  - Some HW (notably PS3) has built-in normalize()
    - Usually beneficial to stick to normalize() there
- reflect(i, n) = i – 2.0f * dot(i, n) * n
- lerp(a, b, c) implemented as (b-a) * c + a
  - If c and either a or b are scalar, b * c + a * (1-c) is fewer ops

Instead of normalize(), you could roll a normfactor() function that computes the scalar normalizing factor. Any other scalar factor that needs to go in there could then be multiplied into this factor before the final multiply with the vector.

Double-check with PS3 if you support this platform as it has a built-in normalize() that could be faster, depending on lots of factors such as the phase of the moon and whether you passed any virgin blood on the command-line.

Hidden scalar math

- 50.0f * normalize(vec) = 50.0f * (vec * rsqrt(dot(vec, vec)))
  - Unnecessarily doing vector math

```
float3 main(float3 vec : TEXCOORD0) : SV_Target
{
    return 50.0f * normalize(vec);
}
```

```
0  x: DOT4_e   ____,  R0.x,  R0.x
   y: DOT4_e   ____,  R0.y,  R0.y
   z: DOT4_e   ____,  R0.z,  R0.z
   w: DOT4_e   ____,  (0x80000000, -0.0f).x, 0.0f
1  t: RSQ_e    ____,  PV0.x
2  x: MUL_e    ____,  R0.y,  PS1
   y: MUL_e    ____,  R0.x,  PS1
   w: MUL_e    ____,  R0.z,  PS1
3  x: MUL_e    R0.x,  PV2.y, (0x42480000, 50.0f).x
   y: MUL_e    R0.y,  PV2.x, (0x42480000, 50.0f).x
   z: MUL_e    R0.z,  PV2.w, (0x42480000, 50.0f).x
```

```
float3 main(float3 vec: TEXCOORD) : SV_Target
{
    return vec * (50.0f * rsqrt(dot(vec, vec)));
}
```

```
0  x: DOT4_e   ____,  R0.x,  R0.x
   y: DOT4_e   ____,  R0.y,  R0.y
   z: DOT4_e   ____,  R0.z,  R0.z
   w: DOT4_e   ____,  (0x80000000, -0.0f).x, 0.0f
1  t: RSQ_e    ____,  PV0.x
2  w: MUL_e    ____,  PS1,  (0x42480000, 50.0f).x
3  x: MUL_e    R0.x,  R0.x,  PV2.w
   y: MUL_e    R0.y,  R0.y,  PV2.w
   z: MUL_e    R0.z,  R0.z,  PV2.w
```

The straightforward way of making a vector be length 50.0f is to normalize it and then multiply by 50.0f, which unfortunately is also slower than necessary. This illustrates the benefit of separating the scalar and vector parts of an expression.

Hidden common sub-expressions

- normalize(vec) and length(vec) contain dot(vec, vec)
  - Compiler reuses exact matches
  - Compiler does NOT reuse different uses
- Example: Clamping vector to unit length

```
float3 main(float3 v : TEXCOORD0) : SV_Target
{
    if (length(v) > 1.0f)
        v = normalize(v);
    return v;
}
```

```
dp3  r0.x, v0.xyzx, v0.xyzx
sqrt r0.y, r0.x
rsq  r0.x, r0.x
mul  r0.xzw, r0.xxxx, v0.xxyz
lt   r0.y, l(1.000000), r0.y
movc o0.xyz, r0.yyyy, r0.xzwx, v0.xyzx
```

```
0  x: DOT4_e       ____,  R0.x,  R0.x
   y: DOT4_e       R1.y,  R0.y,  R0.y
   z: DOT4_e       ____,  R0.z,  R0.z
   w: DOT4_e       ____,  (0x80000000, -0.0f).x,  0.0f
1  t: SQRT_e       ____,  PV0.x
2  w: SETGT_DX10   R0.w,  PS1,  1.0f
   t: RSQ_e        ____,  R1.y
3  x: MUL_e        ____,  R0.z,  PS2
   y: MUL_e        ____,  R0.y,  PS2
   z: MUL_e        ____,  R0.x,  PS2
4  x: CNDE_INT     R0.x,  R0.w,  R0.x,  PV3.z
   y: CNDE_INT     R0.y,  R0.w,  R0.y,  PV3.y
   z: CNDE_INT     R0.z,  R0.w,  R0.z,  PV3.x
```

Here is another example. The dot-product is shared, because the sub-expressions match. However, the compiler doesn't take advantage of the mathematical relationship between sqrt(x) and rsqrt(x).

Hidden common sub-expressions

- Optimize: Clamping vector to unit length

Original

```
if (length(v) > 1.0f)
    v = normalize(v);
return v;
```

Expand expressions

```
if (sqrt(dot(v, v)) > 1.0f)
    v *= rsqrt(dot(v, v));
return v;
```

Unify expressions

```
if (rsqrt(dot(v, v)) < 1.0f)
    v *= rsqrt(dot(v, v));
return v;
```

```
float norm_factor =
    min(rsqrt(dot(v, v)), 1.0f);
v *= norm_factor;
return v;
```

Extract sub-exp and flatten

```
float norm_factor =
    saturate(rsqrt(dot(v, v)));
return v * norm_factor;
```

Replace clamp with saturate

```
precise float norm_factor =
    saturate(rsqrt(dot(v, v)));
return v * norm_factor;
```

HLSL compiler workaround

The most obvious optimization, i.e. removing the sqrt() call and comparing the length squared instead, is a bit of a dead-end. We get further by unifying the expressions instead. Once the expressions are unified, we can pull out the normalizing factor, and then simply flatten the if-statement though clamping the factor to 1.0f. As we don't expect any negative numbers, this clamp can be replaced with saturate(). Finally, HLSL realizes as much too, so we need to apply the precise workaround.

Hidden common sub-expressions

- Optimize: Clamping vector to unit length

```
float3 main(float3 v : TEXCOORD0) : SV_Target
{
    if (length(v) > 1.0f)
        v = normalize(v);
    return v;
}
```

```
0  x: DOT4_e      ____, R0.x, R0.x
   y: DOT4_e     R1.y, R0.y, R0.y
   z: DOT4_e      ____, R0.z, R0.z
   w: DOT4_e      ____, (0x80000000, -0.0f).x, 0.0f
1  t: SQRT_e      ____, PV0.x
2  w: SETGT_DX10 R0.w, PS1,  1.0f
   t: RSQ_e       ____, R1.y
3  x: MUL_e       ____, R0.z, PS2
   y: MUL_e       ____, R0.y, PS2
   z: MUL_e       ____, R0.x, PS2
4  x: CNDE_INT    R0.x, R0.w, R0.x, PV3.z
   y: CNDE_INT    R0.y, R0.w, R0.y, PV3.y
   z: CNDE_INT    R0.z, R0.w, R0.z, PV3.x
```

```
float3 main(float3 v : TEXCOORD0) : SV_Target
{
    if (rsqrt(dot(v, v)) < 1.0f)
        v *= rsqrt(dot(v, v));
    return v;
}
```

```
0  x: DOT4_e      ____, R0.x, R0.x
   y: DOT4_e      ____, R0.y, R0.y
   z: DOT4_e      ____, R0.z, R0.z
   w: DOT4_e      ____, (0x80000000, -0.0f).x, 0.0f
1  t: RSQ_e       ____, PV0.x
2  x: MUL_e       ____, R0.y, PS1
   y: MUL_e       ____, R0.x, PS1
   z: SETGT_DX10  ____, 1.0f, PS1
   w: MUL_e       ____, R0.z, PS1
3  x: CNDE_INT    R0.x, PV2.z, R0.x, PV2.y
   y: CNDE_INT    R0.y, PV2.z, R0.y, PV2.x
   z: CNDE_INT    R0.z, PV2.z, R0.z, PV2.w
```

Unifying expressions basically only removed the sqrt() call. Which is not bad of course, it even saved a VLIW instruction slot here. About the same as the simple optimization of comparing the square length. The main advantage of this route is that it allows us to go further with more optimizations. The key point is that the rsqrt() has to be computed anyway, so we can take advantage of its existence and design the if-statement on what is already available.

Hidden common sub-expressions

- Optimize: Clamping vector to unit length

```
float3 main(float3 v : TEXCOORD0) : SV_Target
{
    precise float norm_factor =
        saturate(rsqrt(dot(v, v)));
    return v * norm_factor;
}
```

```
0  x: DOT4_e ____, R0.x, R0.x
   y: DOT4_e ____, R0.y, R0.y
   z: DOT4_e ____, R0.z, R0.z
   w: DOT4_e ____, (0x80000000, -0.0f).x,  0.0f
1  t: RSQ_e  ____, PV0.x      CLAMP
2  x: MUL_e  R0.x, R0.x, PS1
   y: MUL_e  R0.y, R0.y, PS1
   z: MUL_e  R0.z, R0.z, PS1
```

- Extends to general case
  - Clamp to length 5.0f → norm_factor = saturate(5.0f * rsqrt(dot(v, v)));

Once we have gone all the way through we have a really short stub left of the original code. This code is also easily extended to a more general case, clamping to any given length, and that only adds a single scalar multiply, whereas it would have added at least three in the naïve implementation.

The general case is of course more useful for real tasks, such as for instance clamping a motion vector for motion blur to avoid over-blurring some fast moving objects, something we did in Just Cause 2 for the main character. The main takeaway here though is that understanding what happens inside of built-in functions allows us to write better code, and even built-ins should be scrutinized for splitting scalar and vector work.

Evaluation order

- Expressions evaluated left-to-right
  - Except for parentheses and operator precedence
  - Place scalars to the left and/or use parentheses

```
//     float3    float     float   float3       float  float
return Diffuse * n_dot_l * atten * LightColor * shadow * ao;
```

```
0  x: MUL_e      ____,  R0.z,  R0.w
   y: MUL_e      ____,  R0.y,  R0.w
   z: MUL_e      ____,  R0.x,  R0.w
1  y: MUL_e      ____,  R1.w,  PV0.x
   z: MUL_e      ____,  R1.w,  PV0.y
   w: MUL_e      ____,  R1.w,  PV0.z
2  x: MUL_e      ____,  R1.x,  PV1.w
   z: MUL_e      ____,  R1.z,  PV1.y
   w: MUL_e      ____,  R1.y,  PV1.z
3  x: MUL_e      ____,  R2.x,  PV2.w
   y: MUL_e      ____,  R2.x,  PV2.x
   w: MUL_e      ____,  R2.x,  PV2.z
4  x: MUL_e      R2.x,  R2.y,  PV3.y
   y: MUL_e      R2.y,  R2.y,  PV3.x
   z: MUL_e      R2.z,  R2.y,  PV3.w
```

```
//     float3    float3     (float     float   float  float)
return Diffuse * LightCol * (n_dot_l * atten * shadow * ao);
```

```
0  x: MUL_e      R0.x,  R0.x,  R1.x
   y: MUL_e      ____,  R0.w,  R1.w
   z: MUL_e      R0.z,  R0.z,  R1.z
   w: MUL_e      R0.w,  R0.y,  R1.y
1  x: MUL_e      ____,  R2.x,  PV0.y
2  w: MUL_e      ____,  R2.y,  PV1.x
3  x: MUL_e      R0.x,  R0.x,  PV2.w
   y: MUL_e      R0.y,  R0.w,  PV2.w
   z: MUL_e      R0.z,  R0.z,  PV2.w
```

Unfortunately, this optimization opportunity frequently goes unnoticed, but it is one of the best and most general applicable optimizations. It benefits all hardware, and even more so on the most modern ones. Definitively look out for this one on PC and next-gen platforms, but even vector based architectures such as curr-gen consoles typically see a nice improvement as well. And it's all just about simple rearrangement of the code that normally doesn't affect readability at all.

Evaluation order

- VLIW & vector architectures are sensitive to dependencies
  - Especially at beginning and end of scopes
  - a * b * c * d = ((a * b) * c) * d;
    - Break dependency chains with parentheses: (a*b) * (c*d)

```
//     float     float   float  float  float3    float3
return n_dot_l * atten * shadow * ao * Diffuse * LightColor;
```

```
0  x: MUL_e      ____,  R0.w,  R1.w
1  w: MUL_e      ____,  R2.x,  PV0.x
2  z: MUL_e      ____,  R2.y,  PV1.w
3  x: MUL_e      ____,  R0.y,  PV2.z
   y: MUL_e      ____,  R0.x,  PV2.z
   w: MUL_e      ____,  R0.z,  PV2.z
4  x: MUL_e      R1.x,  R1.x,  PV3.y
   y: MUL_e      R1.y,  R1.y,  PV3.x
   z: MUL_e      R1.z,  R1.z,  PV3.w
```

```
//      float     float     float  float    float3    float3
return (n_dot_l * atten) * (shadow * ao) * (Diffuse * LightColor);
```

```
0  x: MUL_e      ____,  R2.x,  R2.y
   y: MUL_e      R0.y,  R0.y,  R1.y      VEC_021
   z: MUL_e      R0.z,  R0.x,  R1.x      VEC_120
   w: MUL_e      ____,  R0.w,  R1.w
   t: MUL_e      R0.x,  R0.z,  R1.z
1  w: MUL_e      ____,  PV0.x,  PV0.w
2  x: MUL_e      R0.x,  R0.z,  PV1.w
   y: MUL_e      R0.y,  R0.y,  PV1.w
   z: MUL_e      R0.z,  R0.x,  PV1.w
```

This is for VLIW and vector architectures. It doesn't help scalar based hardware, but it doesn't hurt them either. They are just not affected.

What we are doing here is basically just breaking up the dependency chain into a “tree” if you will, basically allowing more parallelism. The number of operations doesn't change at all, but the required instruction slots is reduced, which will result in faster execution.

Real-world testing

- Case study: Clustered deferred shading
  - Mixed quality code
    - Original lighting code quite optimized
    - Various prototype quality code added later
  - Low-level optimization
    - 1-2h of work
    - Shader about 7% shorter
    - Only sunlight: 0.40ms → 0.38ms (5% faster)
    - Many pointlights: 3.56ms → 3.22ms (10% faster)
  - High-level optimization
    - Several weeks of work
    - Between 15% slower and 2x faster than classic deferred
    - Do both!

High-level optimization, i.e. changing the algorithm, tends to have a greater impact. Nothing new there. They also tend to be vastly more costly in terms of time and effort. The ROI of low-level optimizations tends to be far greater. But this is not an argument for or against either, because you should do both if you aspire to have any sort of technical leadership.

The preferable way is of course not to go stomping on all the shaders in your code base looking for low-level optimizations. That's fine say at the end of a project, or when you need to poke around in shader anyway. What you really should do is design your high-level algorithm fully aware of the hardware, and have a low-level thinking as you're writing the shader to begin with. Don't just check in what happened to work first, but make sure you've covered at least the most obvious low-level optimizations before submitting anything to production.

Additional recommendations

- Communicate intention with [branch], [flatten], [loop], [unroll]
  - [branch] turns “divergent gradient” warning into error
    - Which is great!
    - Otherwise pulls chunks of code outside branch
- Don't do in shader what can be done elsewhere
- Move linear ops to vertex shader
  - Unless vertex bound of course
- Don't output more than needed
  - SM4+ doesn't require float4 SV_Target
  - Don't write unused alphas!

```
float2 ClipSpaceToTexcoord(float3 Cs)
{
    Cs.xy = Cs.xy / Cs.z;
    Cs.xy = Cs.xy * 0.5h + 0.5h;
    Cs.y = ( 1.h - Cs.y );
    return Cs.xy;
}
```

↓

```
float2 tex_coord = Cs.xy / Cs.z;
```

The [branch] tag is one of the best features in HLSL. If you intend to skip some work for performance where applicable, always apply the tag to communicate this to the compiler. Because if the compiler fails to do it, there will be an error that you can fix. Otherwise it will silently flatten the branch, slowing down the shader rather than speeding it up, and you may not even notice. And while in this state, chances are that more branch-unfriendly code will be added that you will have to fix later.

How can I be a better low-level coder?

- Familiarize yourself with GPU HW instructions
  - Also learn D3D asm on PC
- Familiarize yourself with HLSL ↔ HW code mapping
  - GPUShaderAnalyzer, NVShaderPerf, fxc.exe
  - Compare result across HW and platforms
- Monitor shader edits' effect on shader length
  - Abnormal results? → Inspect asm, figure out cause and effect
  - Also do real-world benchmarking

For Just Cause 2 we made a shader diff script that basically showed the changes an edit did to the number of instructions and registers used by the shader. Especially when you have something like an über-shader with many specializations it allowed us to catch cases where a change had impacts on versions that were expected to be unaffected. You could also get a great overview of the impact of updating a function in a central header file used by everything and see an instruction or two shaved off from loads of shaders in the project. We made it a standard practice to attach the diff to code-reviews that affected shaders, allowing us to also judge the performance impact on new features or other changes, as well as staying on top of general shader code quality.

Optimize all the shaders!

References

[1] IEEE-754

[2] Floating-Point Rules

[3] Fabian Giesen: Finish your derivations, please

Questions?

@_Humus_
emil.persson@avalanchestudios.se

We are hiring!
New York, Stockholm

AVALANCHE STUDIOS

Join our team!