



Dark Secrets of Shader Development or What Your Mother Never Told You About Shaders

Overview

- What are shaders?
 - Shader compilation process
- Shader optimizations
 - Non-hardware specific shader optimizations
 - Hardware specific shader optimizations for ATI
 - Vertex shader optimizations
 - Pixel shader optimizations



What Are Shaders?

- Shaders are micro-programs controlling vertex and pixel engines in the graphics hardware
- In DirectX® shaders are streams of tokens sent to hardware through API
 - Tokens are specially coded assembly instructions
 - Drivers receive shaders in their genuine form
 - DirectX® shaders are special pseudo code (p-code)
 - Drivers receive macros just like any other instructions



What Do Drivers Do With Shaders?

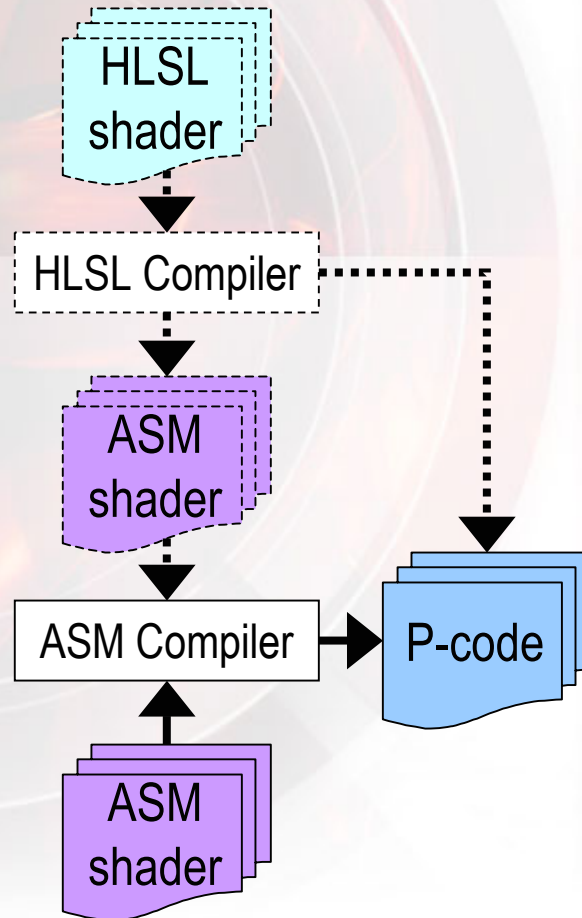
- DirectX® shaders don't exactly correspond to the hardware shader implementation
- Drivers compile for specific hardware platform into special hardware microcode (μ -code)
- Drivers optimize shaders for specific platform
 - Carefully designed shaders allow drivers to optimize shader code more efficiently for specific hardware



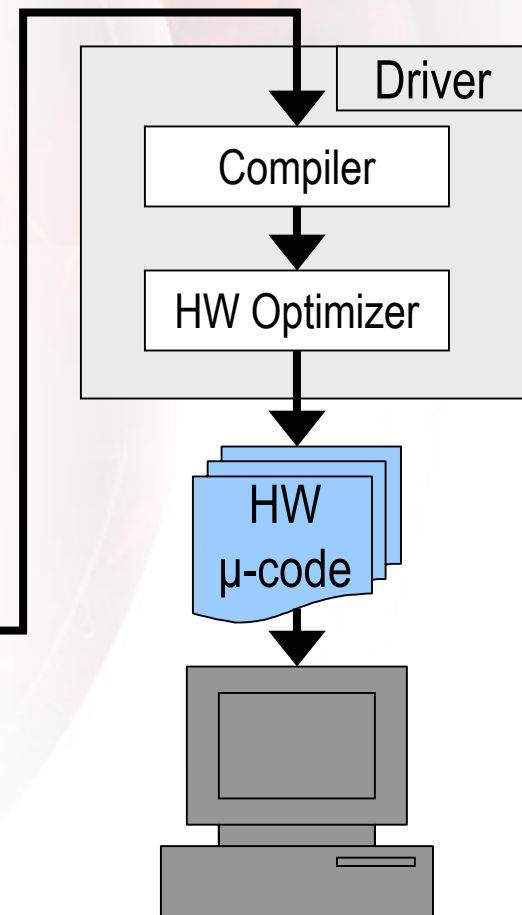
Shader Compilation Process



Front-end Compiler



Back-end Compiler



Shader at Various Compilation Stages

HLSL code: `float x=(a*abs(f))>(b*abs(f));`

Assembly
code: `abs r7.w, c2.x
mul r2.w, r7.w, c0.x
mul r9.w, r7.w, c1.x
slt r0.w, r9.w, r2.w`

DirectX®
p-code:

02000023	80080007	a0000002	
03000005	80080002	80ff0007	a0000000
03000005	80080009	80ff0007	a0000001
0300000c	d00f0000	80ff0009	80ff0002

Hardware
specific
µ-code:

983a28f41595
329d8d123c04
329d8d627c08
3b487794333a



Shader Optimizations

- Optimization – process of making something as perfect, functional or effective as possible
 - Make development process as efficient as possible
 - Make shaders perform as fast as possible
 - Make rendered images as perfect looking as possible
- Optimization is an art of balancing performance, image quality and efforts it takes to makes it “perfect”



Shader Optimizations

- Use tools like RenderMonkey for development and visual debugging
- Use HLSL and concentrate efforts on algorithmic optimizations
- Use lower level optimizations specific for shader processors (i.e. vectorize calculations)
 - Requires good understanding of hardware and knowledge of assembly
- Use hardware specific optimizations whenever necessary to get the most out of hardware



Optimization #1 (HLSL) Use Intrinsic Functions

- Don't reinvent the wheel, use intrinsic functions
 - Code uses all available hardware features
 - Optimized code for each shader model



Without Optimization

```
float MyDOT(float3 v1,  
            float3 v2)  
{  
    return (v1.x * v2.x +  
            v1.y * v2.y +  
            v1.z * v2.z);  
}
```

...

```
float v = MyDOT(N, L);
```

...

With Optimization

...

```
float v = dot(N, L);
```

...

Optimization #1 (HLSL) Use Intrinsic Functions

- Assembly translation of HLSL code...



Without Optimization

```
vs_2_0  
. . .  
mul r0.xy, v0, c0  
add r0.w, r0.y, r0.x  
mad oPos, c0.z, v0.z, r0.w  
. . .
```

With Optimization

```
vs_2_0  
. . .  
dp3 r0.w, v0, c1  
. . .
```

Optimization #2 (HLSL)

Properly Use Data Types

- Use the most appropriate data types in calculations (`float`, `float2`, `float3` and `float4`)
 - Don't use vector in place of scalar calculations
 - Don't use `float4` where you could use `float3`
- This optimization allows HLSL compiler and/or driver optimizer to pair shader instructions whenever possible



Optimization #3 (HLSL)

Reduce Typecasting

- Get rid of typecasting when it's not needed
 - Indirectly initialize unused vector components (i.e. alpha channel)

Without Optimization

```
sampler texMap;  
float4 diff, amb;  
  
float4 main(float2 t:  
    TEXCOORD0) : COLOR  
{  
    float3 color =  
        tex2D(texMap, t);  
    color *= diff + amb;  
    return float4(color, 0);  
}
```

With Optimization

```
sampler texMap;  
float4 diff, amb;  
  
float4 main(float2 t:  
    TEXCOORD0) : COLOR  
{  
    float4 color =  
        tex2D(texMap, t);  
    color *= diff + amb;  
    return color;  
}
```




Optimization #3 (HLSL)

Reduce Typecasting

- And this is how it translates to assembly...

Without Optimizations



```
ps_2_0
def c2, 0, 0, 0, 0
dcl t0.xy
dcl_2d s0
texld r0, t0, s0
mov r1.xyz, c1
add r7.xyz, c0, r1
mul r2.xyz, r7, r0
mov r2.w, c2.x
mov oC0, r2
```

With Optimizations

```
ps_2_0

dcl t0.xy
dcl_2d s0
texld r0, t0, s0
mov r1, c1
add r7, c0, r1
mul r2, r7, r0
mov oC0, r2
```


Optimization #4 (HLSL)

Avoid Integer Calculations

- Instead of integers rely on floats for math
- HLSL supports integer arithmetic, but most hardware doesn't
- Compiler emulates `int` type support
 - Precision and range might vary
 - Some extra code is generated

Without Optimization

```
float4 main(int k :  
    TEXCOORD0) : COLOR  
{  
    int n = k / 3;  
  
    return n;  
}
```

With Optimizations

```
float4 main(float k :  
    TEXCOORD0) : COLOR  
{  
    float n = k / 3;  
  
    return n;  
}
```



Optimization #4 (HLSL)

Avoid Integer Calculations

- Assembly code confirms inefficiency...



Without Optimization

```
ps_2_0
def c0, 0.333333, 1, 0, 0
dcl t0.x
mul r0.w, t0.x, c0.x
frc r7.w, r0.w
cmp r9.w, -r7.w, c0.w, c0.y
add r4.w, r0.w, -r7.w
cmp r11.w, r0.w, c0.w, c0.y
mad r1, r9.w, r11.w, r4.w
mov oC0, r1
```

With Optimization


```
ps_2_0
def c0, 0.333333, 0, 0, 0
dcl t0.x
mul r0, t0.x, c0.x
mov oC0, r0
```

Optimization #5 (HLSL)

Use Integers For Indexing

- When indexing into arrays of constants use integers instead of floats

Without Optimization



```
float4x4 m[10];

float4 main(
    float4 p: POSITION,
    float2 ind: BLENDINDICES,
    float blend: BLENDWEIGHT)
: POSITION
{
    float4 p1 =
        mul(p, m[ind.x]);
    float4 p2 =
        mul(p, m[ind.y]);
    return lerp(p1, p2,
        blend);
}
```

With Optimizations

```
float4x4 m[10];


float4 main(
    float4 p: POSITION,
    int2 ind: BLENDINDICES,
    float blend:
        BLENDWEIGHT) : POSITION
{
    float4 p1 =
        mul(p, m[ind.x]);
    float4 p2 =
        mul(p, m[ind.y]);
    return lerp(p1, p2,
        blend);
}
```

Optimization #5 (HLSL)

Use Integers For Indexing

- Assembly shows that floats add rounding

Without Optimization



```
vs_2_0
def c40, 4, 0, 0, 0
dcl_position v0
dcl_blendindices v1
dcl_blendweight v2
frc r0.xy, v1
add r0.xy, -r0, v1
mul r0.xy, r0, c40.x
mov a0.xy, r0
dp4 r0.x, v0, c0[a0.y]
dp4 r0.y, v0, c1[a0.y]
dp4 r0.z, v0, c2[a0.y]
dp4 r0.w, v0, c3[a0.y]
dp4 r1.x, v0, c0[a0.x]
dp4 r1.y, v0, c1[a0.x]
dp4 r1.z, v0, c2[a0.x]
dp4 r1.w, v0, c3[a0.x]
add r0, r0, -r1
mad oPos, v2.x, r0, r1
```

With Optimization

```
vs_2_0
def c40, 4, 0, 0, 0
dcl_position v0
dcl_blendindices v1
dcl_blendweight v2
mul r0.xy, v1, c40.x
mov a0.xy, r0
dp4 r0.x, v0, c0[a0.y]
dp4 r0.y, v0, c1[a0.y]
dp4 r0.z, v0, c2[a0.y]
dp4 r0.w, v0, c3[a0.y]
dp4 r1.x, v0, c0[a0.x]
dp4 r1.y, v0, c1[a0.x]
dp4 r1.z, v0, c2[a0.x]
dp4 r1.w, v0, c3[a0.x]
add r0, r0, -r1
mad oPos, v2.x, r0, r1
```

Optimization #6 (HLSL, ASM)

Pack Scalar Constants

- Combine scalar constants into full vectors
 - Reduces number of constants
 - Allows to work around hardware limitations (read-port limit)

Without Optimization

```
float scale, bias;
```

```
float4 main(float4 Pos :  
    POSITION) : POSITION  
{  
    return (Pos *  
        scale +  
        bias);  
}
```

With Optimization

```
float2 scale_bias;
```

```
float4 main(float4 Pos :  
    POSITION) : POSITION  
{  
    return (Pos *  
        scale_bias.x +  
        scale_bias.y);  
}
```



Optimization #6 (HLSL, ASM)

Pack Scalar Constants

- Here's assembly version of this optimization



Without Optimization

```
vs_2_0  
dcl_position v0  
mul r0, v0, c0.x  
add oPos, r0, c1.x
```

With Optimization

```
vs_2_0  
dcl_position v0  
mad oPos, v0, c0.x, c0.y
```

Optimization #7 (HLSL)

Pack Arrays of Constants

- Pack array elements into full constant vectors
 - Similar to previous optimization tip

Without Optimization

```
float fArray[8];

float4 main(float4 v:
    COLOR) : COLOR
{
    float a = 0;
    int i;
    for(i = 0; i < 8; i++)
    {
        a += fArray[i];
    }
    return v * a;
}
```

With Optimization

```
float4 fPackedArray[2];
static float fArray[8] =
    (float[8]) fPackedArray;

float4 main(float4 v:
    COLOR) : COLOR
{
    float a = 0;
    int i;
    for(i = 0; i < 8; i++)
    {
        a += fArray[i];
    }
    return v * a;
}
```




Optimization #8 (HLSL)

Properly Declare Constants

- For conditional compilation use boolean constants declared as **static**

Without Optimization



```
float4 a;  
bool b = true;  
  
float4 main(  
    float4 i0: TEXCOORD0,  
    float4 i1 : TEXCOORD1) :  
    COLOR  
{  
    if (b)  
        return i0+a;  
    else  
        return i1+a;  
}
```

With Optimization

```
float4 a;  
static bool b = true;  
  
float4 main(  
    float4 i0: TEXCOORD0,  
    float4 i1 : TEXCOORD1) :  
    COLOR  
{  
    if (b)  
        return i0+a;  
    else  
        return i1+a;  
}
```

Optimization #8 (HLSL)

Properly Declare Constants

- Assembly shows that when not declared as **static** both branches are evaluated



Without Optimization

```
ps_2_0
dcl t0
dcl t1
add r1, t0, c0
add r0, t1, c0
cmp r0, -c1.x, r0, r1
mov oC0, r0
```

With Optimization

```
ps_2_0
dcl t0
add r0, t0, c0
mov oC0, r0
```

Optimization #9 (HLSL, ASM)

Vectorize Calculations

- Whenever possible vectorize code by joining similar operations together
 - It's possible to perform up to 4-x operations in one shot

Without Optimization

```
float4 main(float k: COLOR)
: COLOR
{
    float a, b, c, d;
    a = k + 1;
    b = k + 2;
    c = k + 3;
    d = k + 4;
    return
        float4(a, b, c, d);
}
```

With Optimization

```
float4 main(float k: COLOR)
: COLOR
{
    float4 v;
    v = k + float4(1,2,3,4);
    return v;
}
```




Optimization #9 (HLSL, ASM)

Vectorize Calculations

- The same optimization in assembly code...

Without Optimization



```
ps_2_0
def c0, 1, 2, 3, 4
dcl v0.x
add r0.x, v0.x, c0.x
add r0.y, v0.x, c0.y
add r0.z, v0.x, c0.z
add r0.w, v0.x, c0.w
mov oC0, r0
```

With Optimization

```
ps_2_0
def c0, 1, 2, 3, 4
dcl v0.x
add r0, v0.x, c0
mov oC0, r0
```

Optimization #10 (HLSL, ASM)

Vectorize Even More

- Use similar approach to take advantage of special instructions available in shaders, i.e. use dot product instructions
 - Example: $a+b+c+d \Leftrightarrow a*1+b*1+c*1+d*1 \Leftrightarrow \text{DP4}$

Without Optimization

```
float a, b, c, d;  
a = k * j;  
b = k + j;  
c = k - j;  
d = j - k;  
a = a + b + c + d;
```

With Optimization

```
float4 v;  
v.x = k * j;  
v.y = k + j;  
v.z = k - j;  
v.w = j - k;  
a = dot(v, 1);
```




Optimization #10 (HLSL, ASM)

Vectorize Even More

- The same optimization in assembly...

Without Optimization



```
...  
mul r0.w, r7.w, r8.w  
add r1.w, r7.w, r8.w  
sub r2.w, r7.w, r8.w  
sub r3.w, r8.w, r7.w  
add r0.w, r0.w, r1.w  
add r0.w, r0.w, r2.w  
add r0.w, r0.w, r3.w
```

With Optimization

```
def c0, 1, 0, 0, 0  
...  
mul r0.x, r7.w, r8.w  
add r0.y, r7.w, r8.w  
sub r0.z, r7.w, r8.w  
sub r0.w, r8.w, r7.w  
dp4 r0.w, r0, c0.x  
...  
...
```

Optimization #11 (HLSL)

Vectorize Comparisons

- Currently conditional operators in HLSL don't properly promote scalars to vectors
 - I.e. implementing `a = (c > 0.5f) ? 0.1f : 0.9f;`

Without Optimization

```
float4 main(float4 c: COLOR)
: COLOR
{
    float4 a;
    a.x = (c.x > 0.5f) ?
        0.1f : 0.9f;
    a.y = (c.y > 0.5f) ?
        0.1f : 0.9f;
    a.z = (c.z > 0.5f) ?
        0.1f : 0.9f;
    a.w = (c.w > 0.5f) ?
        0.1f : 0.9f;
    return a;
}
```

With Optimization

```
float4 main(float4 c:
COLOR) : COLOR
{
    float4 a;
    a = (c >
        float4(0.5f,0.5f,
0.5f,0.5f)) ?
        float4(0.1f,0.1f,
0.1f,0.1f) :
        float4(0.9f,0.9f,
0.9f,0.9f);
    return a;
}
```




Optimization #11 (HLSL)

Vectorize Comparisons

- Vectorized comparison assembly

Without Optimization



```
ps_2_0
def c0, 0.5, 0.1, 0.9, 0
dcl v0
add r1.w, -v0.x, c0.x
add r0.w, -v0.y, c0.x
cmp r0.x, r1.w, c0.y, c0.z
add r1.w, -v0.z, c0.x
cmp r0.y, r0.w, c0.y, c0.z
add r0.w, -v0.w, c0.x
cmp r0.z, r1.w, c0.y, c0.z
cmp r0.w, r0.w, c0.y, c0.z
mov oC0, r0
```

With Optimization

```
ps_2_0
def c0, 0.5, 0.1, 0.9, 0
dcl v0
add r1, -v0, c0.x
cmp r0, r1, c0.y, c0.z
mov oC0, r0
```


Optimization #12 (HLSL)

Careful With Matrix Transpose

- Avoid transposing matrices in HLSL code
- Use reversed **mul** () operand order for multiplication by transposed matrix
- Use column order matrices whenever possible

Without Optimization

```
float3x4 m;
```

```
float4 main(float4 p:  
    POSITION) : POSITION  
{  
    float3 v;  
    v = mul(m, p);  
    return float4(v, 1);  
}
```

With Optimization

```
float4x3 m;
```

```
float4 main(float4 p:  
    POSITION) : POSITION  
{  
    float3 v;  
    v = mul(p, m);  
    return float4(v, 1);  
}
```



Optimization #12 (HLSL)

Careful With Matrix Transpose

- Column order matrix transformation takes 3 **DP4** instructions vs. 4 **MUL/MAD**



Without Optimization

```
vs_2_0
def c4, 1, 0, 0, 0
dcl_position v0
mul r0.xyz, v0.x, c0
mad r2.xyz, v0.y, c1, r0
mad r4.xyz, v0.z, c2, r2
mad oPos.xyz, v0.w, c3, r4
mov oPos.w, c4.x
```

With Optimization

```
vs_2_0
def c3, 1, 0, 0, 0
dcl_position v0
m4x3 oPos.xyz, v0, c0
mov oPos.w, c3.x
```

Optimization #13 (PS:HLSL,ASM)

Use Swizzles Wisely

- PS 2.0 doesn't support arbitrary swizzles
 - Creatively use existing swizzles (i.e. .WZYX gives you access to .ZW in reversed order)
 - Can be used for constant packing and vectorization

Without Optimization

```
float2 a, b, c, d;
sampler s;

float4 main(float2 i0:
    TEXCOORD0, float2 i1 :
    TEXCOORD1) : COLOR
{
    float4 j =
        tex2D(s, (i0+a)*c) +
        tex2D(s, (i1+b)*d);
    return j;
}
```

With Optimization

```
float4 ab, cd;
sampler s;

float4 main(float4 i01 :
    TEXCOORD0) : COLOR
{
    float4 t = (i01+ab)*cd;
    float4 j =
        tex2D(s, t.xy) +
        tex2D(s, t.wz);
    return j;
}
```




Optimization #13 (PS:HLSL,ASM)

Use Swizzles Wisely

- Optimization in assembly...
 - Using .ZW instead of .WZ would produce two **MOV** instructions instead of one

Without Optimization



```
ps_2_0
dcl t0.xy
dcl t1.xy
dcl_2d s0
add r0.xy, t0, c0
mul r0.xy, r0, c2
add r1.xy, t1, c1
mul r1.xy, r1, c3
texld r0, r0, s0
texld r1, r1, s0
add r0, r0, r1
mov oC0, r0
```

With Optimization

```
ps_2_0
dcl t0
dcl_2d s0
add r0, t0, c0
mul r0, r0, c1
mov r1.xy, r0.wzyx
texld r0, r0, s0
texld r1, r1, s0
add r0, r0, r1
mov oC0, r0
```

Optimization #14 (PS: HLSL)

Use 1D Texture Fetches

- In DirectX® 1D textures are emulated by 2D textures of $N \times 1$ dimensions
- For fetching 1D textures use special intrinsic function `tex1D` even though texture is really 2D

Without Optimization

```
sampler texMap;  
float3 L;  
  
float4 main(float3 V :  
    TEXCOORD0) : COLOR  
{  
    float2 t = 0;  
    t.x = dot(V, L);  
    return tex2D(texMap, t);  
}
```

With Optimization

```
sampler texMap;  
float3 L;  
  
float4 main(float3 V :  
    TEXCOORD0) : COLOR  
{  
    float t;  
    t.x = dot(V, L);  
    return tex1D(texMap, t);  
}
```



Optimization #14 (PS: HLSL) Use 1D Texture Fetches

- Equivalent code in assembly ...



Without Optimization

```
ps_2_0
def c1, 0, 0, 0, 0
dcl t0.xyz
dcl_2d s0
dp3 r0.x, t0, c0
mov r0.y, c1.x
texld r7, r0, s0
mov oC0, r7
```

With Optimization

```
ps_2_0

dcl t0.xyz
dcl_2d s0
dp3 r0.xy, t0, c0
texld r7, r0, s0
mov oC0, r7
```

Optimization #15 (PS: HLSL,ASM) Use Signed Textures

- Use signed texture formats to represent signed data (i.e. normal maps)
- PS 2.0 doesn't support **_bx2** modifier, so it takes extra **MAD** instruction to expand range of unsigned data



ATI Hardware Specific Optimizations

- Optimizations for DirectX® 9 members of RADEON family
 - RADEON 9500, 9600, 9700, 9800
- Vertex shader optimizations
- Pixel shader optimizations
- Precision in pixel shaders



Vertex Shader Optimizations For RADEON 9500+

- Only a few optimization rules apply
 - Drivers do all the dirty work for you
- Most relevant vertex shader optimizations for ATI DirectX® 9 hardware
 - Vertex data output from shaders
 - Instruction co-issue
 - Use of flow control



ATI VS Optimization #1

Vertex Data Output

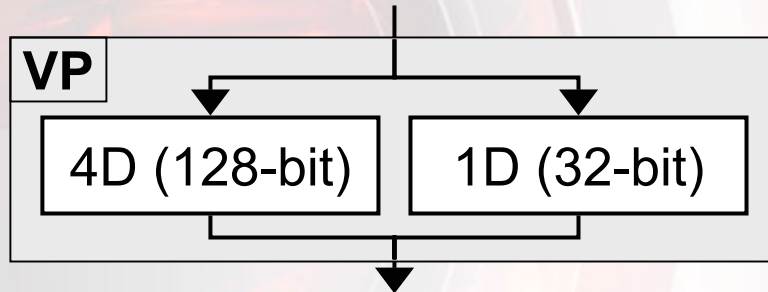
- Export computed vertex position as early as possible
 - Driver does a good job of helping you with that
 - Shortest chain of instructions computing vertex position allows optimizer to do its job
- Output from shader only necessary information
 - Don't output texture coordinates with the same data
 - Use PS 1.4 and PS 2.0 for mapping texture coordinates to samplers



ATI VS Optimization #2

Instruction Co-issue

- Vertex processor architecture allows co-issuing vertex shader instructions (4D+1D per clock), somewhat similar to pixel shaders



- Follow these rules to increase chance of co-issue:
 - Don't use scalar computations to write to output registers
 - Use write mask (.w) in **POW**, **EXP**, **LOG**, **RCP** and **RSQ** instructions
 - Remember that read port limits apply to instruction pair

ATI VS Optimization #3

Flow Control

- VS 2.0 supports constant based static branching and loops
 - Allows to simplify shader management and reduces number of shaders
 - The flow control instructions come for “free”
 - Drivers optimize flow control execution
- Performance of shaders with flow control might be reduced due to the limited scope of shader optimizations



Pixel Shader Optimizations For RADEON 9500+

- Optimized pixel shaders considerably increase graphics performance
 - Driver allows to get the most out of carefully designed shader
- Most important pixel shader optimizations for ATI graphics hardware
 - Texture instructions
 - Dependent texture reads
 - ALU instructions
 - Instruction balancing
 - Instruction co-issue
 - Use of PS 1.4



ATI PS Optimization #1

TEXKILL Instruction

- Avoid using **TEXKILL** whenever possible
 - Don't use **TEXKILL** for user clip planes if clipping can be done at vertex level
- Truth about **TEXKILL** instruction on ATI's hardware:
 - Positioning of **TEXKILL** in shader code doesn't affect performance
 - Shaders don't have early out ability
 - **TEXKILL** instruction affects "top of the pipe Z-reject" efficiency
 - **TEXKILL** is a texture instruction, it contributes to creation of levels of dependency in the shader



ATI PS Optimization #1

Dependency Levels And TEXKILL

- Example of creating extra level of dependency in the pixel shader with **TEXKILL**
 - Note that first ALU instructions without texture reads count as dependency level in this example



```
ps_2_0  
def c0, 0.3, 1, 0.5, 0
```

```
sub r0, t0, c0
```

```
texkill r0
```

```
mov r1, v0
```

```
mov oC0, r1
```

- 1st level

- 2nd level

ATI PS Optimization #2

Depth Value Computations

- Limit cases where depth value is output from pixel shaders
 - PS 1.4 – **TEXDEPTH** instruction
 - PS 2.0 – output to **oDepth** register
- Truth about outputting depth value from pixel shaders
 - Affects Hyper-Z efficiency
 - Interferes with “top of the pipe Z-reject”



ATI PS Optimization #3

Dependent Texture Read


- Keep in mind that dependent texture reads aren't free
 - 1-2 levels are executed at top performance
 - 3-4 levels are executed slower, but performance is still reasonable for practical use
- Try spreading instructions equally across levels of dependency
- Avoid adding extra levels of dependency



ATI PS Optimization #3

Dependent Texture Read

- Example of creating unnecessary dependent texture reads



```
ps_2_0
dcl_2d s0
dcl t0.xy
add r0.xy, t0, c0
texld r0, r0, s0
mul r1, r0, c8
add r0.xy, t0, c1
texld r0, r0, s0
mad r1, r0, c9, r1
add r0.xy, t0, c2
texld r0, r0, s0
mad r1, r0, c10, r1
mov oC0, r1
```

- 1st level

- 2nd level

```
ps_2_0
dcl_2d s0
dcl t0.xy
add r0.xy, t0, c0
add r1.xy, t0, c1
add r2.xy, t0, c2
texld r0, r0, s0
mul r3, r0, c8
texld r1, r1, s0
mad r3, r1, c9, r3
texld r2, r2, s0
mad r3, r2, c10, r3
mov oC0, r3
```

- 3rd level

- 4th level

ATI PS Optimization #4

Arithmetic Instructions

- All simple instructions execute at the rate of 1 instruction/clock in each pipe
- Most of the macros execute as described in DirectX® 9 documentation
 - Macro **SINCOS** takes up as many as 8 clocks
- There's no penalty for immediately reusing computed result
 - Don't try to be too "smart" with reordering instructions to hide inexistent latencies

```
. . .  
sub r0, r0, c0  
mad r0, r0, r1, c1  
mul r0, r0, r0  
. . .
```

◀= 1 instruction per clock



ATI PS Optimization #5

Instruction Balancing

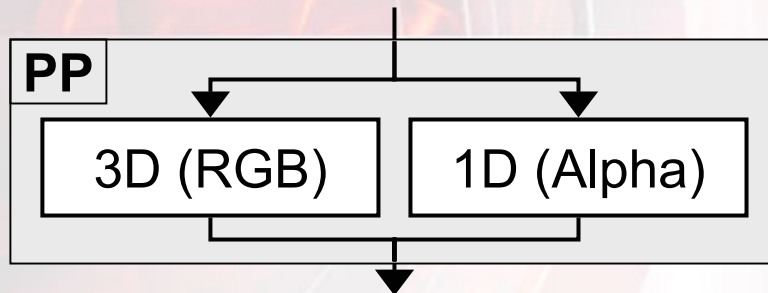
- Pixel engines can simultaneously read textures and perform ALU operations!
- When texture bandwidth isn't a bottleneck try to keep number of texture instructions close to number of arithmetic instructions
- Can use function map lookups to balance texture vs. ALU instructions
- However, always keep in mind image quality



ATI PS Optimization #6

Instruction Co-issue

- PS 2.0 model doesn't support instruction pairing
- RADEON 9500+ is still build around “dual-pipe” design



- In PS 2.0 on RADEON 9500+ it's still possible to pair vector calculations (RGB-pipe) with scalar operations (Alpha-pipe) to be executed on the same cycle



ATI PS Optimization #6

Instruction Co-issue

- Driver optimizes for co-issue
- Rules for taking advantage of automatic instruction pairing:
 - Spread the computational load between color and alpha pipes
 - PS 2.0 doesn't have explicit pairing; use write masks (.rgb and .a) for automatic co-issue in the driver
 - Use scalar instructions **RCP**, **RSQ**, **EXP** and **LOG** only in alpha pipe
 - Make it easier for optimizer to find co-issued instructions by placing them close to each other in the code



ATI PS Optimization #6

Instruction Co-issue

- Example: calculation of diffuse and specular lighting

Without Optimization

```
...  
dp3 r0, r1, r0      // N.H  
dp3 r2, r1, r2      // N.L  
mul r2, r2, r3      // * color  
mul r2, r2, r4      // * tex  
mul r0.r, r0.r, r0.r // spec^2  
mul r0.r, r0.r, r0.r // spec^4  
mul r0.r, r0.r, r0.r // spec^8  
mad r0.rgb, r0.r, r5, r2  
...
```

Total: 8 instructions

With Optimization

```
...  
dp3 r0, r1, r0      // N.H  
dp3 r2.r, r1, r2     // N.L  
mul r6.a, r0.r, r0.r // spec^2  
mul r2.rgb, r2.r, r3 // * color  
mul r6.a, r6.a, r6.a // spec^4  
mul r2.rgb, r2, r4    // * tex  
mul r6.a, r6.a, r6.a // spec^8  
mad r0.rgb, r6.a, r5, r2  
...
```

Total: 5 instructions



ATI PS Optimization #7

Using PS 1.4

- PS 2.0 doesn't expose many instruction and operand modifiers
- RADEON 9500+ architecture still supports many old modifiers
- Use PS 1.4 to get access to modifiers in cheap shaders that require a lot of modifiers

ps_1_4

. . .

dp3_d4 r0, r0_bx2, r1_bx2

. . .

ps_2_0

def c0, 2, -1, 0.25, 0

. . .

mad r0, r0, c0.z, c0.y

mad r1, r1, c0.z, c0.y

dp3 r0, r0, r1

mul r0, r0, c0.z

. . .



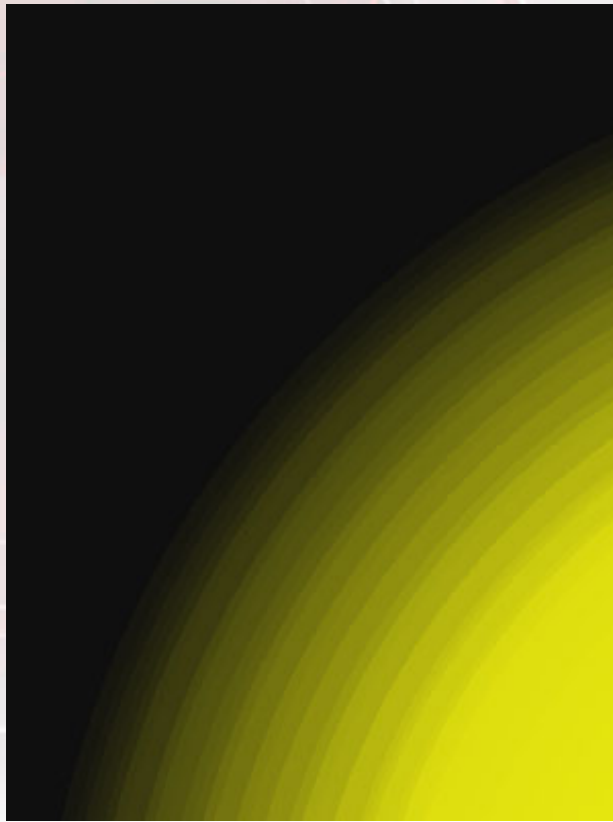
Precision In Pixel Shaders

- In RADEON 9500+ all pixel calculations happen in 24-bit float format (s7e16)
- ATI hardware doesn't support partial precision mode by design
 - Insufficient precision when working with texture coordinates
 - Lower quality when computing reflections, specular lighting and procedural textures
- Optimizations should take into account image quality especially when cinematographic quality is a goal

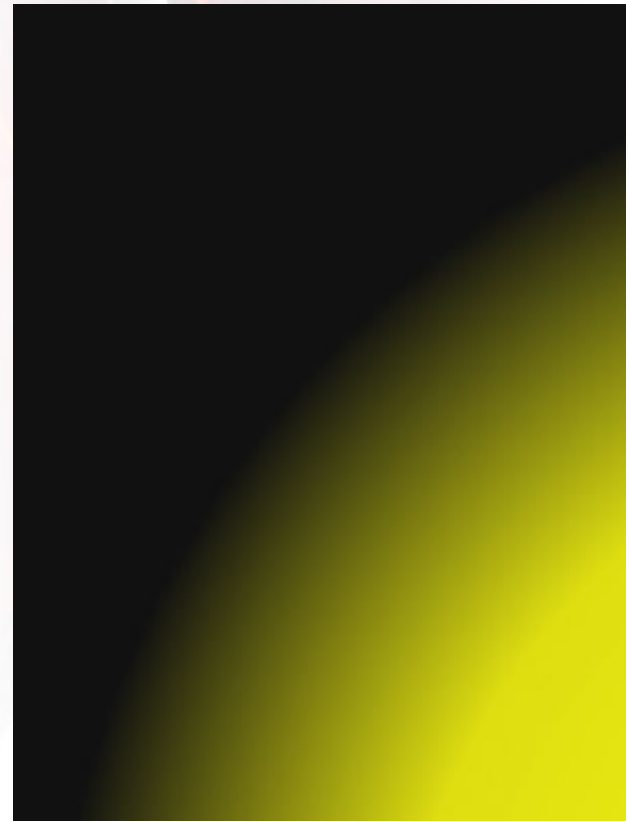


Examples of Precision In Pixel Shaders

- Point light source



16-bit precision

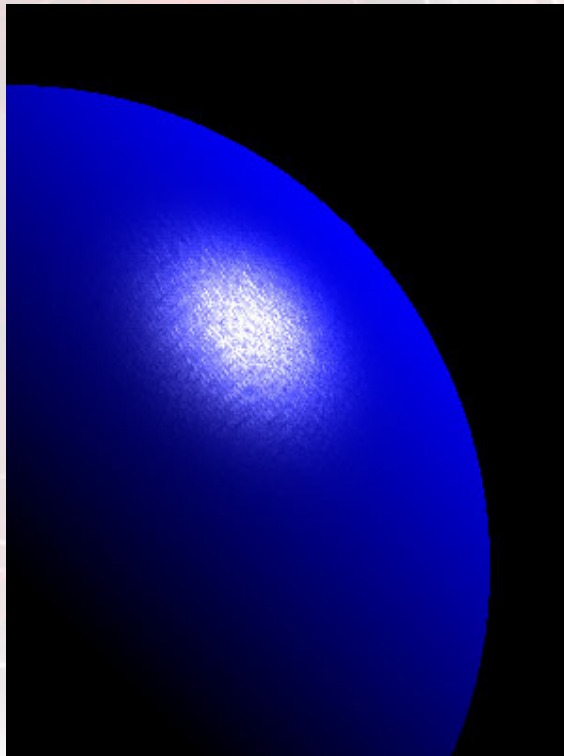


24-bit precision

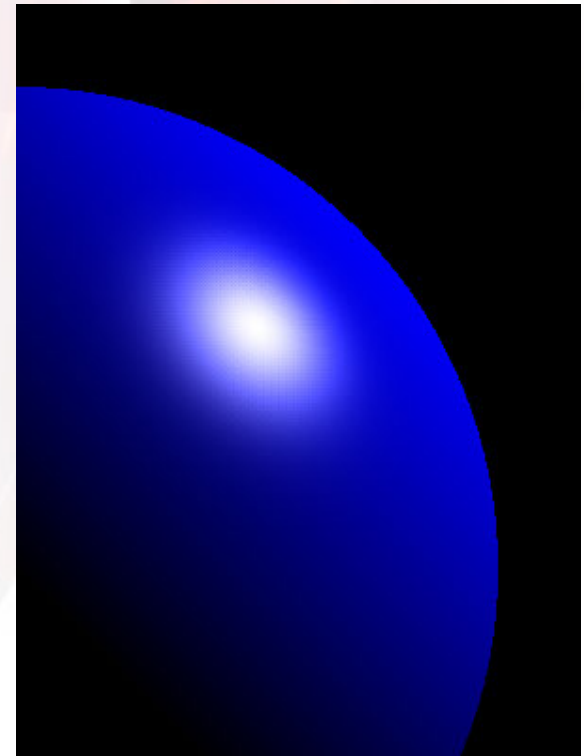


Examples of Precision In Pixel Shaders

- Normal vector normalization in pixel shaders: cubemap vs. **NRM** instruction



256x256 cubemap



NRM instruction

