MICHAL.DROBOT

3D @ UBISOFT MTL

KRAKOW TECHNOLOGY PARK

DIGITAL DRAGONS

DIGITAL DRAGONS
8-9 MAJA 2014
STARA ZAJEZDNIA / KRAKÓW, POLSKA

LOW LEVEL OPTIMIZATIONS FOR GCN

Hacking The New Generation

2014-05-11

Agenda

- GCN Architecture
- Instruction Set
- Optimizing ALU
  - New Ways
  - Ideas
  - Tips & Tricks
- Use Cases

GCN Architecture

- **AMD Radeon HD 7xxx R7x R9x**
- **Microsoft Xbox One**
- **Sony Playstation 4**

GCN Architecture

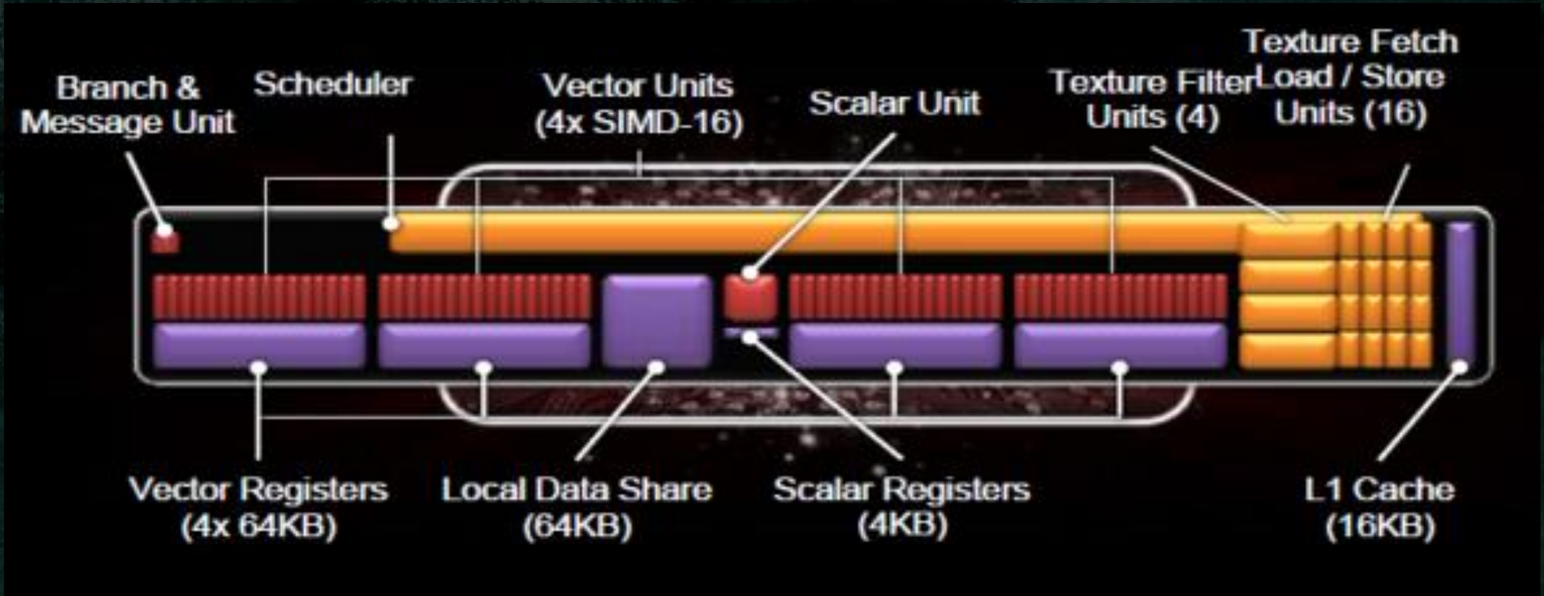
- Open documentation by AMD
  - ISA

- Well covered basics
  - „Low-level Shader Optimization for Next-Gen and DX11” – Emil Persson
  - „The AMD GCN Architecture: A Crash Course" - Layla Mah

- Basics
  - Keep it Wide : Occupancy : Low Resource (VGPR) usage
  - Lots of ‚smart’ ALU – trade for bandwidth
  - GCN super good at hiding latency – but needs help

Optimizing ALU

- GCN CU can execute 256 SP Vector ALU in 4 clk
- Each lane dispatches 1 SP ALU op / clk
- Each SP ALU takes 4 clk
- SQ can dispatch from different wavefront each clk



Basic OP Performance

- 32bit arithmetic and logical OPS
  - Full Rate
  - Exception int32 mul/mad @Quarter Rate
    - Use int24 mul/mad if applicable @Full Rate
- 64bit arthimetic and logical OPS
  - Half Rate
  - Exception mul/mad @Quarter Rate
- Conversion and Packing OPS
  - All operands <=32bit @Full Rate
  - Any operand >32bit @Half Rate

Special OP Performance

- **Transcendental Functions**
  - Use linear approximations
    - Single function runs at all (4) SIMDs simultaniously
  - @Quarter Rate
  - Rcp, Sqrt, rsqrt, log, pow, exp, sin, cos
  - Supporting OPs
    - Cleanup, accuracy, denormal flushing @Full Rate

- **32 bit graphics special OP**
  - @Full Rate
  - CubeMap OPs, Packed Byte OPs

Macro OP Performance

- Macro unrolled OP
  - tan, div, atan, acos, asin
  - Smoothstep
  - Length
  - normalize
- Unroll into IEEE compliant approximations
  - Very expensive
- Integer DIV
  - Emulated with FP math
  - Multiple Full and Quarter Rate OPs

Code Flow Performance : BRANCH

- **BRANCH**
  - >= 4 WAIT states ~ >= 16 cycles ~ >= 16 Full Rate
    - Branch support and logic
  - Additional latency due to potential I$ miss
  - Additioanl VGPR usage for IC / Label
  - Can skip all OPS
    - Much faster than BRANCH or SELECT
  - Use always in case of redundant Buffer/Texture Memory OP
    - Branch Latency <= L1$ latency

Code Flow Performance : VSKIP

- **VSKIP**
  - Special control flow mode
    - Skips Vector OPs at rate of 10 wavefronts / clk
    - CAN NOT VSKIP VMEM Ops
  - For small pieces of Vector only code
    - Much faster than BRANCH or SELECT
  - Compilers are still catching up
    - Write direct ASM if allowed

Code Flow Performance : SELECT

- **SELECT**
  - **Standard selector**
  - **Execute two code paths**
    - **SELECT one result based on Comparison**
  - **[flatten]**
  - **Ternary Logical Operator**
  - **CndMask()**

```
// SELECT use case
[flatten]
if(x > y)
	z = x;
else
	z = y;
/////////////
z = x > y ? x : y; // v_cndmask_b32 @ Full Rate
```

VOP3 – 3 Operand Vector Instructions

- **IEEE flags free instructions banks for modifiers**
  - **Input Modifiers :**
    - **abs() neg()**
  - **Output Modifiers:**
    - **mul2 mul4 mul8 div2 div4**
    - **saturate()**

```
// Using VOP3
Out.x = saturate(abs(inV.x) * (-inV.y) * 4.0);
```

```
// In one OP
v_mul_f32 v0, abs(s), -v0 mul:4 clamp
```

VOP3 – 3 Operand Vector Instructions

- Most compilers will automatically use VOP3 when
  - Allowed (-fastmath –IEEEStrcit disabled)
  - (-x)
  - Saturate()
  - *2.0 *4.0 *8.0 *0.5 *0.25

VOP3 – Restrictions

- VOP3(VDST, VSRC0, VSRC1, VSRC2)
  - VSRC0 – literals , VGPR, SGPR
  - VSRC1 / VSRC2 – some restrictions on certain combinations
- i.e. Can not issue both VSRC1 and VSRC2 from SGRPRs

- Forces suboptimal
  - SGRP to VGRP preload
  - Disables VOP3

VOP3 – Restrictions - Example

```
float2 TexcoordToScreenPos(float2 inUV, float4 inFov)
{
	float2 p = inUV;
	p.x		= p.x *  inFov.x + inFov.z;
	p.y		= p.y *  inFov.y + inFov.w;
	return p;
}
```

```
s_buffer_load_dwordx4 s[0:3], s[12:15], 0x08
s_waitcnt     lgkmcnt(0)
v_mov_b32     v2, s2
v_mov_b32     v3, s3
s_waitcnt     vmcnt(0) & lgkmcnt(15)
v_mac_f32     v2, s0, v0
v_mac_f32     v3, s1, v1
```

VOP3 – Restrictions – Example - Patch

```
float2 TexcoordToScreenPos(float2 inUV)
{
       float2 p = inUV;
       p.x          = p.x *  2.0 + (- 1.0);
       p.y          = p.y * -2.0 +    1.0;
       return p;
}
```

```
v_mad_f32     v0, v0, 2.0, -1.0
v_mad_f32     v1, v1, -2.0, 1.0
```

VOP3 – Constant Patching

- Built In Fast Literal Constants
  - Built in +/- 1 , 2 ,4, 8 constants - can use with all VOP3

- Single Literal Constant support
  - v_madak_f32
  - v_madmk_f32

VOP3 – Constant Patching

- Consider Uniform Patching
  - If uniforms are constant
    - Artist Generated shaders
    - Particle Systems
    - Transforms

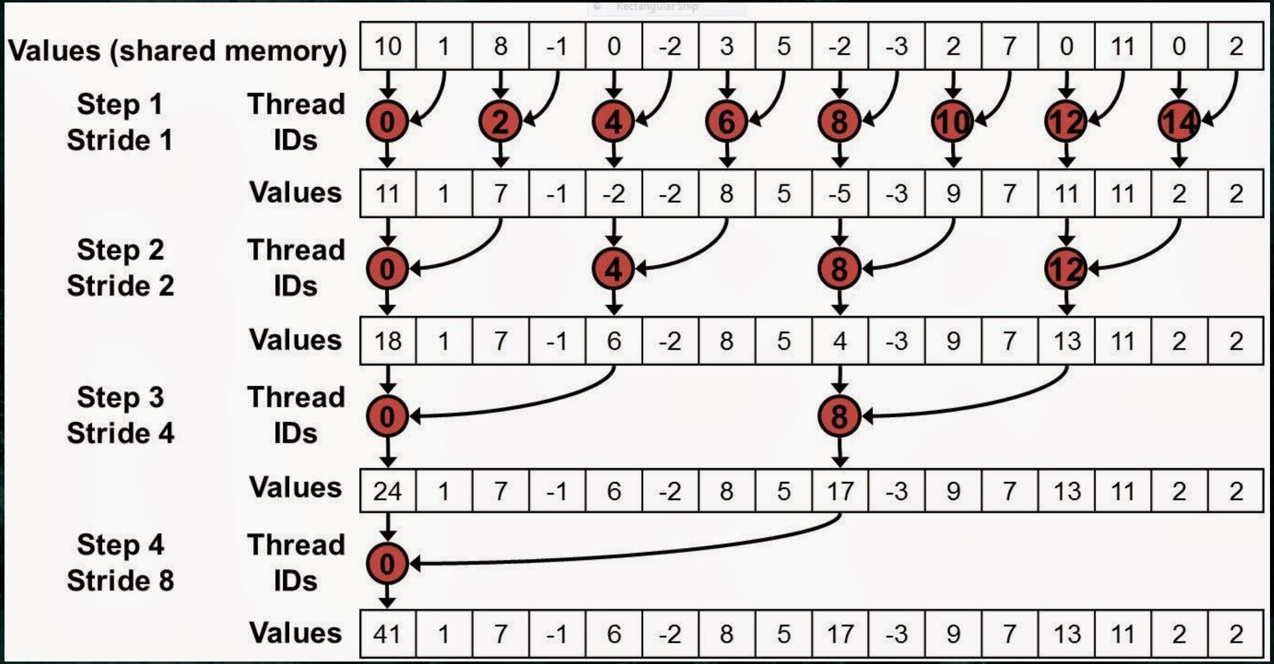
- Beneficial
  - Higher chance of better scheduling
  - Efficient tight loops (i.e. Screen Space Raymarching)

- Balance unique shaders vs performance
  - Crucial shaders can always be patched ‚on flight' – PS3 style

Special ALU OPs : Reduction

- Min3
- Max3
- Med3
  - Median
  - Clamp()

- Optimized
  - Filtering
  - Sorting



Special ALU OPs : Packing

- GCN exposes multiple packing and conversion operations (used for compressed MRT)
  - F32 -> F16
  - F16 -> F32
  - F32 -> SNorm / UNorm
  - ...
  - Also pairwise : 2xF32 -> 2xF16
  - v_cvt_*   - ISA OPs

- Unpacking functions needs to be written manually

Special ALU OPs : BFE

- GCN has full 32bit UINT / INT support
  - Special OPs for masking, shifts, integer artihmetics

- v_bfe_i32
  - BitFieldExtract with sign extension to handle integer based packing
    - Avoids manual care for sign extension due to 2-compliment Integer format
- v_bfe_u32
  - BitFieldExtract to handle unsigned integer based packing
    - Bitmasks, flags, shift + mask

```
// reference implementation for v_bfe_u32
uint BitFieldExtract(uint inSrc, uint inOffset, uint inSize)
{
    return (inSrc >> inOffset) & ((1 << inSize) - 1)
}
```

Special ALU OPs : BFE

```
// reference implementation for v_bfe_i32
int BitFieldExtractSignExtend(int inSrc, uint inOffset, uint inSize)
{
    uint size    = inSize & 0x1f;
    uint offset  = inOffset & 0x1f;
    uint data    = inSrc >> offset;
    uint signBit = data & (1 << (size - 1));
    uint mask    = (1<<size) - 1;

    return (-int(signBit)) | (mask & data);
}
```

```
// Pack 127, -1, -126 into RGB 11 11 10
// Integers are 2's complement
// packed_data = 00001111111 11111111111 1110000010
// Unpack R(127)
BFE(packed_data, 21, 11) = 0000 0000 0000 0000 0000 0000 0111 1111
// Unpack B(-126)
BFE(packed_data,  0, 11) = 1111 1111 1111 1111 1111 1111 1000 0010
```

Special ALU OPs : BFE Pack - Unpack

- **Fast Int Packing and Unpacking**

```
// Int16 packing
int PackInt2ToInt(in int inX, in int inY)
{
    return       (clamp(inX, -int(0x8000), 0x7fff) & 0xffff) |
                 (clamp(inY, -int(0x8000), 0x7f00) << 16);
}

int2 UnpackInt2FromInt(in int inPackedInt)
{
    return int2(
        BitFieldExtractSignExtend((int)inPackedInt, uint(0), uint(16)),
        BitFieldExtractSignExtend((int)inPackedInt, uint(16), uint(16)));
}
```

Special ALU OPs : BFE Pack - Unpack

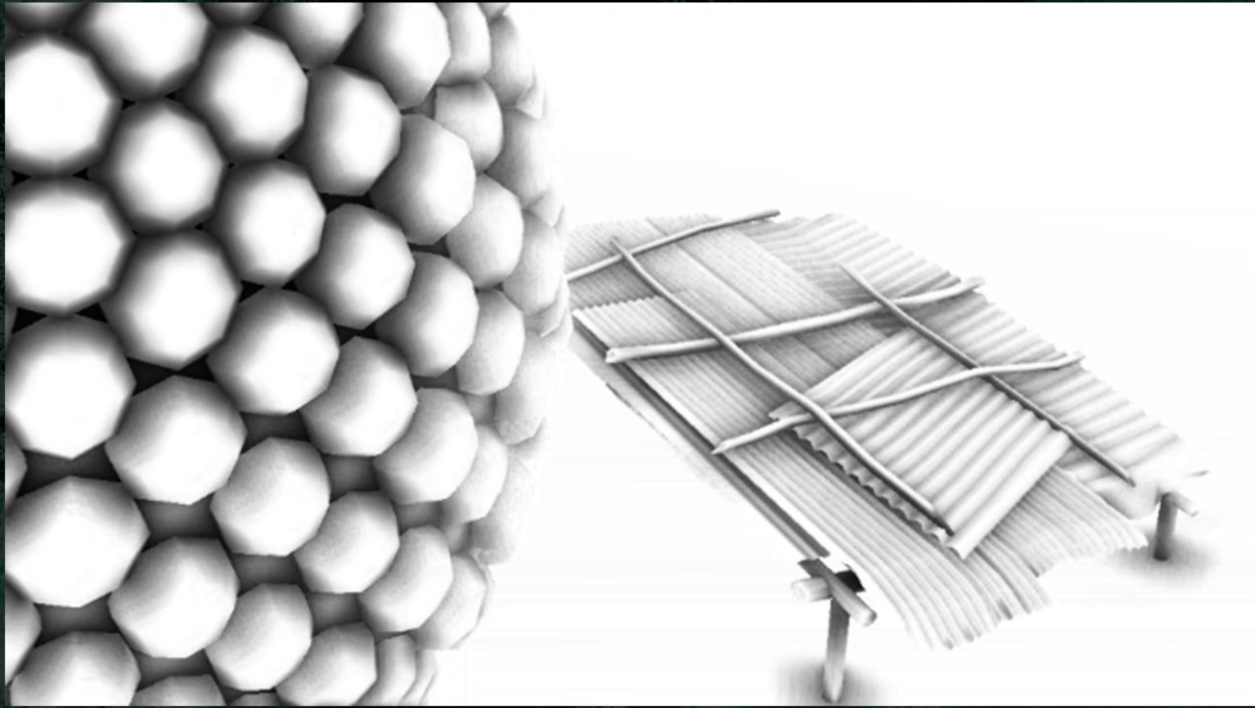
- **Fast SNorm16 Packing - Unpacking**

```
// SNorm16 packing
uint PackSNorm2ToUInt(in float inX, in float inY)
{
    return (clamp(int(inX * 0x7fff), -int(0x7fff), 0x7fff) & 0xffff) |
           clamp(int(inY * 0x7fff), -int(0x7fff), 0x7fff) << 16);
}

float2 UnpackSNorm2FromUInt(in uint inPackedUInt)
{
    return float2(
BitFieldExtractSignExtend((int)inPackedUInt, uint(0) , uint(16))/float(0x7fff),
BitFieldExtractSignExtend((int)inPackedUInt, uint(16), uint(16))/float(0x7fff));
}
```

Special ALU OPs : Sampling Packed Data

- **Pack data into ‚fat' format**
- **Sample with GATHER**

- **Example : Bilateral Filter**
- **Pack all into UINT32**
  - **8 bit DATA**
  - **16 bit Depth**
  - **8 bit Normal**
    - **as 4bit SNORM**
- **1 Gather :**
  - **4 x data**
  - **4 x depth**
  - **4 x normal**



Special ALU OPs : Packing

- **Use Bitfields to reduce register pressure with lifetime bool flags**
  - **Countbits()**
  - **Firstbithigh()**
  - **Firstbitlow()**

```
uint FastLog2(uint inX)
{
    return firstbithigh(inX) - 1;
}
```

Special ALU OPs : BFE – boolean ops

```
// Software Triangle Frustum (near plane) clipping
// Vertex Sorting before line equations
float v0 = b[0]; float v1 = b[1]; float v2 = b[2];
if(b[0] > near_z || b[1] > near_z || b[2] > near_z)
{
    if(b[1] > near_z)    { v0 = t[1]; v1 = t[2]; v2 = t[0]; }
    if(b[2] > near_z)    { v0 = t[2]; v1 = t[0]; v2 = t[1]; }
}
if(
(b[0] > near_z && b[1] > near_z) ||
(b[0] > near_z && b[2] > near_z) ||
(b[1] > near_z && b[2] > near_z)
)
{
    if(!(b[0] > near_z)) { v0 = t[1]; v1 = t[2]; v2 = t[0]; }
    if(!(b[1] > near_z)) { v0 = t[2]; v1 = t[0]; v2 = t[1]; }
}

// Compiles to : 42 ALU @ Full Rate + 4 BRANCH @ (>= 16 FR)
```

Special ALU OPs : BFE – boolean ops optimized

```
// Software Triangle Frustum (near plane) clipping
// Vertex Sorting before line equations
uint bitfield = 0;
bitfield |= b[0] > near_z ? 0x1 << 0 : 0x0;
bitfield |= b[1] > near_z ? 0x1 << 1 : 0x0;
bitfield |= b[2] > near_z ? 0x1 << 2 : 0x0;

float v0 = b[0]; float v1 = b[1]; float v2 = b[2];
uint csb     = CountSetBits(bitfield);
uint csb_eq2 = (csb >> 1) & 0x1;

if(bitfield & 0x2 & csb)           { v0 = t[1]; v1 = t[2]; v2 = t[0]; }
if(bitfield & 0x4 & csb)           { v0 = t[2]; v1 = t[0]; v2 = t[1]; }
if(!(bitfield & 0x1) && csb_eq2) { v0 = t[1]; v1 = t[2]; v2 = t[0]; }
if(!(bitfield & 0x2) && csb_eq2) { v0 = t[2]; v1 = t[0]; v2 = t[1]; }

// Compiles to : 35 ALU @ Full Rate
```

Special ALU Ops : Cubemap

- **Cubemaps are sampled using unified image_sample**
- **Need to calculate face UV and face ID for sampling**
  - **All HW accelerated by custom OPs @Full Rate**

```
v_cubetc_f32  v1, v2, v3, v0 // calculate tc coords
v_cubesc_f32  v4, v2, v3, v0 // calculate tc coords
v_cubema_f32  v5, v2, v3, v0 // calculate major axis
v_cubeid_f32  v8, v2, v3, v0 // calculate face ID
v_rcp_f32     v2, abs(v5)
s_mov_b32     s0, 0x3fc00000
v_mad_f32     v7, v1, v2, s0 // calculate final face UV
v_mad_f32     v6, v4, v2, s0 // calculate final face UV
image_sample  v[0:3], v[6:9], s[4:11], s[12:15] // Tex Array
```

Special ALU Ops : Major Axis

```
// reference implementation for v_cubeid_f32
float CubeMapFaceID(float inX, float inY, float inZ)
{
    float3 v = float3(inX, inY, inZ);
    float faceID;

    if(abs(v.z) >= abs(v.x) && abs(v.z) >= abs(v.y))
    {
        faceID = (v.z < 0.0) ? 5.0 : 4.0;
    }
    else if (abs(v.y) >= abs(v.x))
    {
        faceID = (v.y < 0.0) ? 3.0 : 2.0;
    }
    else
    {
        faceID = (v.x < 0.0) ? 1.0 : 0.0;
    }

    return faceID;
}
```

Special ALU Ops : Major Axis

- Use v_cubeid_f32 , v_cubema_f32 in Major Axis problems
  - Normal Compression
  - Quaternion Compression
  - (Uniform) Custom Kernel Filtering at Cubemap borders
  - Atlased Cubemaps
  - Cubemap raymarching optimizations
  - Several problems in Ray-Casting

Special ALU Ops : Normal Storage Precision

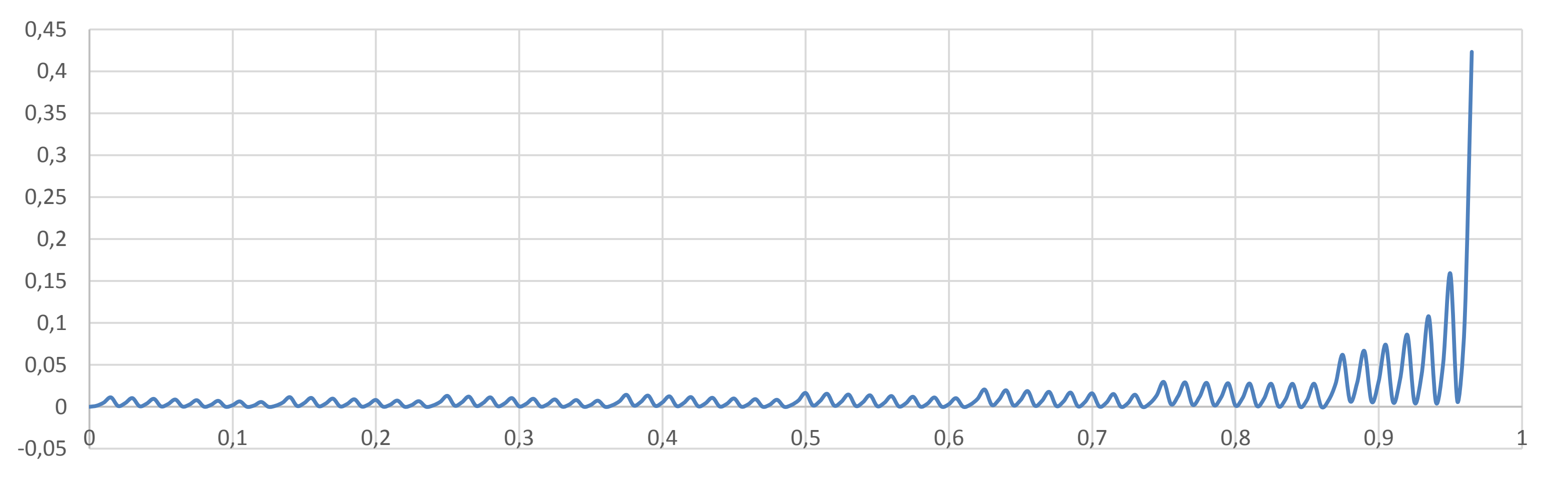
- **Normalized Vector**
- $1 = \sqrt[2]{x^2 + y^2 + z^2}$
- **Store X, Y – reconstruct Z**
- $z = \sqrt[2]{1 - (x^2 + y^2)} = \sqrt[2]{1 - d},\ d = x^2 + y^2$
- **Z precision depends on**
  - $E(z) = dd\ * Er(x, y), where\ E(x)\ is\ error\ function\ of\ storage\ and\ reonstruction$
- $\frac{d}{dd}(z) = \frac{d}{dd}\left(\sqrt{1 - d}\right) = -\frac{1}{2\sqrt{1-d}}$
- **Precision error arises from:**
- $\lim_{d \to 1} -\frac{1}{2\sqrt{1-d}} = \infty$
  - $d = x^2 + y^2 \to 1 \Rightarrow E(z) \to \infty$

Special ALU Ops : Normal Storage Precision

- **Typical way of minimizing error**
  - **Limit the error function by bounds**
- **To minimize E(z) we need to minimize function d**
- **Simple solution**
- $d(x, y) = m^2 + n^2,$
  - $where: \quad m = \min(x, y, z)\,, \quad n = med(x, y, z), \quad 1 = \sqrt[2]{x^2 + y^2 + z^2}$
- $d(x, y)$– **becomes upper bounded by :** $\frac{2}{3}$
- $\lim_{d \to \frac{2}{3}} -\frac{1}{2\sqrt{1-d}} \quad = -\frac{\sqrt{3}}{2}$

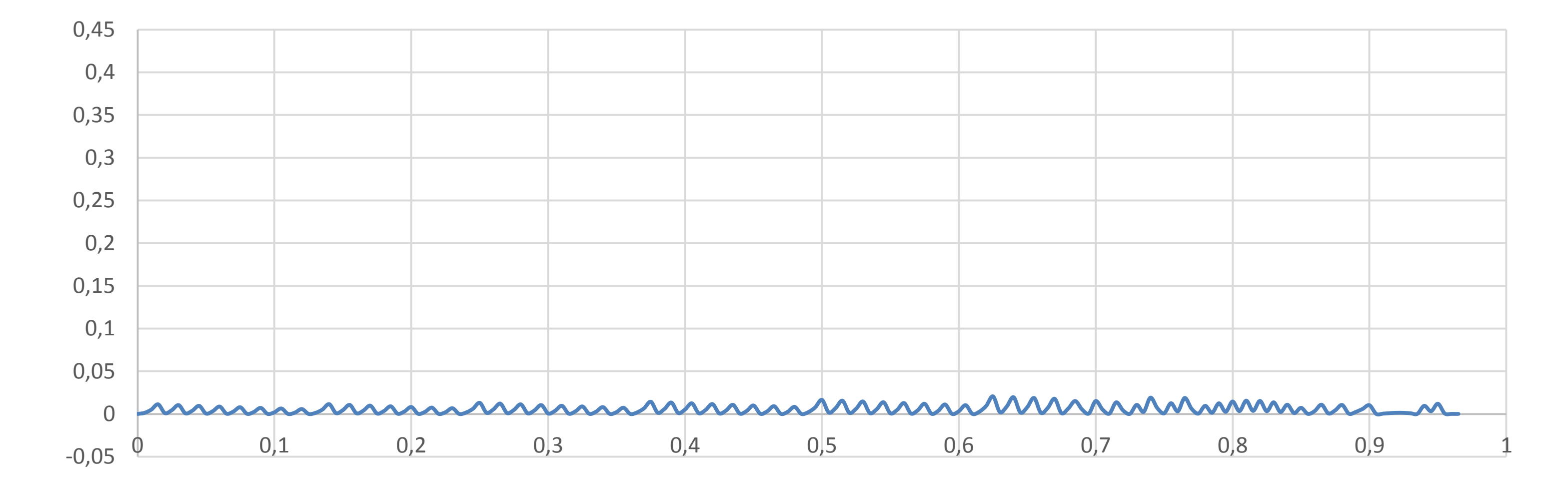
Special ALU Ops : Normal Storage Precision

- $E(n, n') = 1 - n \cdot n'$
- **Standard Reconstruction**
  - **7bit SNorm X, Y + 1bit sign**
  - $MSE(n, n') \approx \frac{3.04}{10000}$, over X, Y domain, where $1 = \sqrt[2]{x^2 + y^2 + z^2}$
  - $Useless\ n' : E(n, n') > \frac{1}{1024} \approx 5.4\%$



Special ALU Ops : Normal Storage Precision

- $E(n, n') = 1 - n \cdot n'$
- **Major Axis (minimum m,n from x,y,z)**
  - **7bit SNorm M,N + 2.5bit sign/order index**
  - $MSE(n, n') \approx \frac{1.18}{10000}$, over X, Y domain, where $1 = \sqrt[2]{x^2 + y^2 + z^2}$
  - $Useless\ n' : E(n, n') > \frac{1}{1024} \approx 0.022\%$



Special ALU Ops : Major Axis

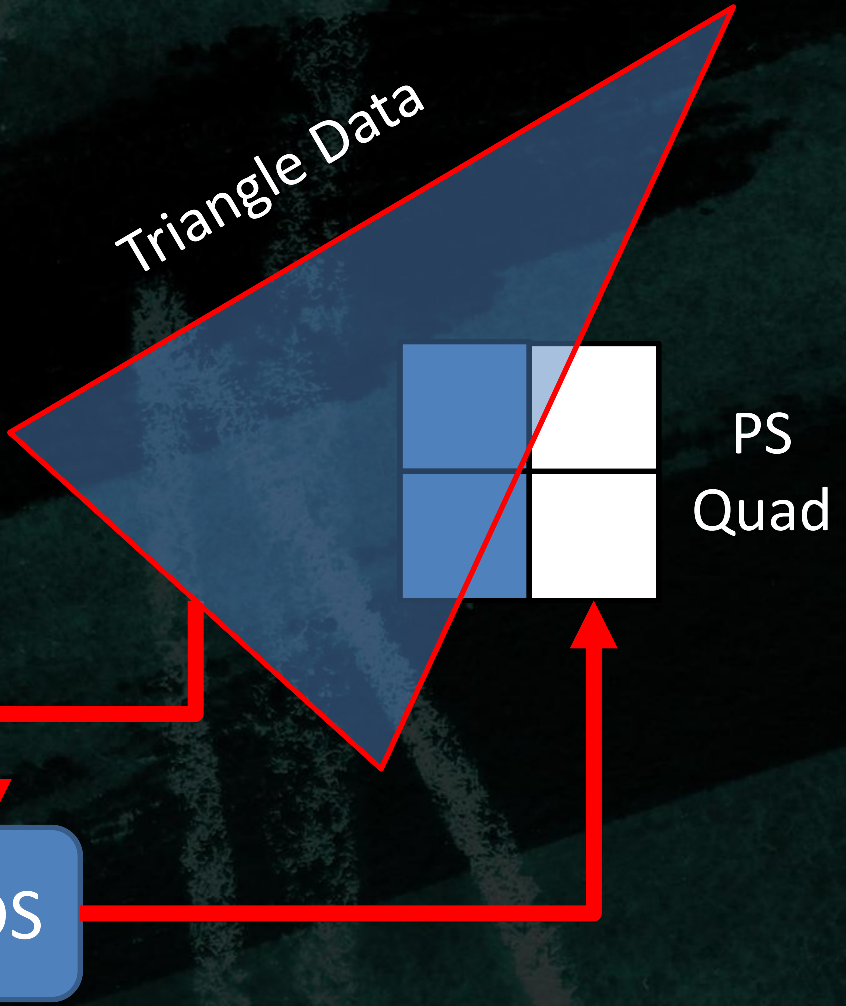
```
float3 PackNormalMajorAxis(float3 inNormal)
{
    uint index = 2;
    if(abs(inNormal.x) >= abs(inNormal.y) && abs(inNormal.x) >= abs(inNormal.z))
            index = 0;
    else if(abs(inNormal.y) > abs(inNormal.z)
        index = 1;

    float3 normal = inNormal;
    normal = index == 0 ? normal.yzx : normal;
    normal = index == 1 ? normal.xzy : normal;

    float s = normal.z > 0.0 ? 1.0 : -1.0;
    float3 packedNormal;
    packedNormal.xy = normal.xy * s;
    packedNormal.z   = index / 2.0f;
    return packedNormal;
}

// Compiles to :
// 28 ALU @ Full Rate + 2 BRANCH @ (>= 16FR)
```

Special ALU Ops : Major Axis

```
float3 PackNormalMajorAxis(float3 inNormal)
{
    uint   index   = CubeMapFaceID(inNormal.x, inNormal.y, inNormal.z) * 0.5f;
    float3 normal = inNormal;

    normal = index == 0 ? normal.yzx : normal;
    normal = index == 1 ? normal.xzy : normal;

    float s = normal.z > 0.0 ? 1.0 : -1.0;
    float3 packedNormal;
    packedNormal.xy = normal.xy * s;
    packedNormal.z   = index / 2.0f;

    return packedNormal;
}

// Compiles to :
// 17 ALU @Full Rate
```

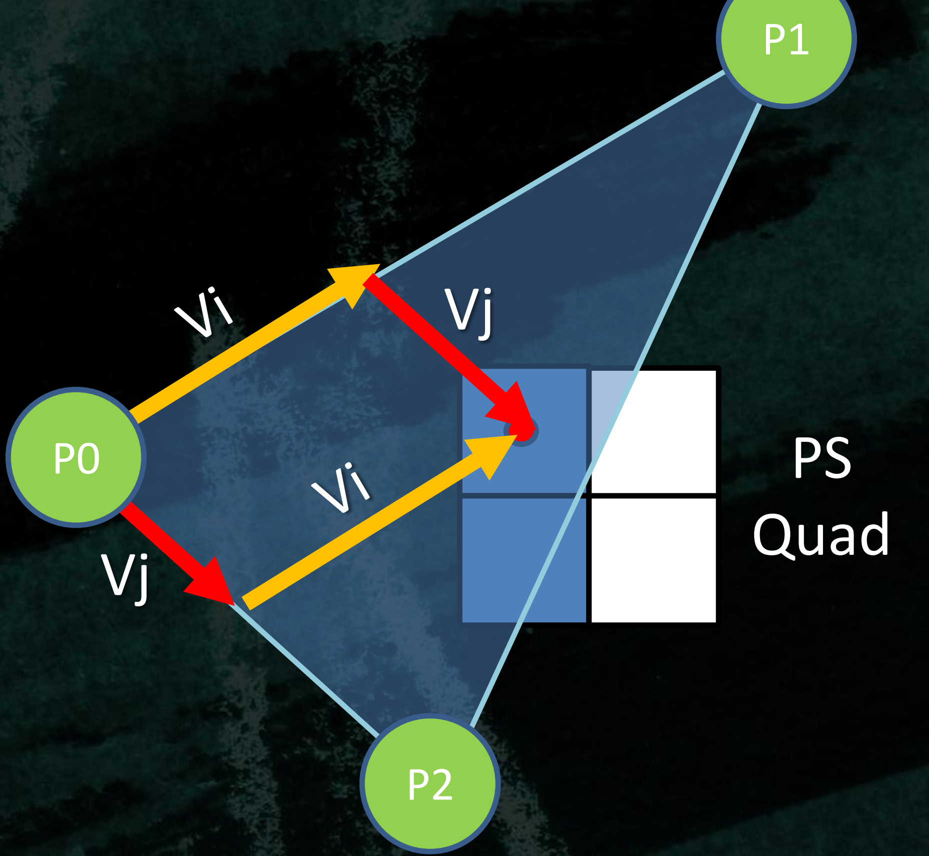
Interpolator : Interpolation

- VS->PS interpolation on GCN is ‚manual'
  - Unrolled by compiler
  - Optimized in HW
- LDS contains vertex data per rasterized traingle
- PS fetches the data and interpolates manually



Interpolator : Interpolation

- **P0, P1, P2**
  - **Hold Vertex Data**
- **Vi Vj**
  - **Barycentric Coordinates**
- **Depending on Interpolant settings**
  - **Interplolate**
    - **At center, sample, centroid**
  - **No interpolation (nointerpolation)**
    - **Also forced on INT type**
    - **Fetches data from V0 – Vertex 0**




```
float4 Interpolate( float4 A, float4 B, float4 C, float2 Vij )
{
   return A * (1.0 - Vij.x - Vij.y) + B * Vij.x + C * Vij.y;
}
```

Interpolator : Mode

```
struct Interpolants
{
    float4 position  : SV_POSITION;
    float4 color     : COLOR0;
};
```

```
float4 main( Interpolants In ) : COLOR{
    float4 Out;
    Out = In.color;
    return Out;
}
```

```
v_interp_p1_f32 v2, v0, attr0.x    // Load Data for Attr0 from LDS and perform
                                   // Vi - first part of
                                   // interpolation (using V00, V01)
v_interp_p2_f32 v2, v1, attr0.x    // Load Data for Attr0 from LDS and perform
                                   // Vj - second part of
                                   // interpolation (using V01, V10)
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
```

Interpolator : Mode

```
struct Interpolants
{
   float4 position  : SV_POSITION;
   int4   color     : COLOR0;
};
```

```
int4 main( Interpolants In ) : COLOR{
   int4 Out;
   Out = n.color;
   return Out;
}
```

```
v_interp_mov_f32 v0, p0, attr0.x  // Load Data from Vertex p0 for Attr0 from LDS
v_interp_mov_f32 v1, p0, attr0.y  // Load Data from Vertex p0 for Attr0 from LDS
v_interp_mov_f32 v2, p0, attr0.z  // Load Data from Vertex p0 for Attr0 from LDS
v_interp_mov_f32 v3, p0, attr0.w  // Load Data from Vertex p0 for Attr0 from LDS
```

Special ALU Ops : Interplator Compression

- GCN allows to poll for HW rasterizer Vi Vj – barycentric coordinates
  - Calculated according to set interpolator flags
- Opens possibility for custom interpolation and packing

- Geometry Shader and Tessellation Pipeline – Data amplfication
  - Requires huge BW
  - Use interpolator packing to optimize BW

- PS can also be bottlenecked by LDS
  - Too much LDS used for ‚fat’ Vertex Data

- Never interpolate triangle const data!

Interpolator : Packing

- **Read Vertex Data**
  - `v_interp_mov_f32 v0, p0,  attr0.x // Vertex P00`
  - `v_interp_mov_f32 v0, p10, attr0.x // Vertex P10`
  - `v_interp_mov_f32 v0, p20, attr0.x // Vertex P20`
- **Barycentric Coordinates Vi Vj**
  - **Preloaded in VGPRs (compiler does it for you)**

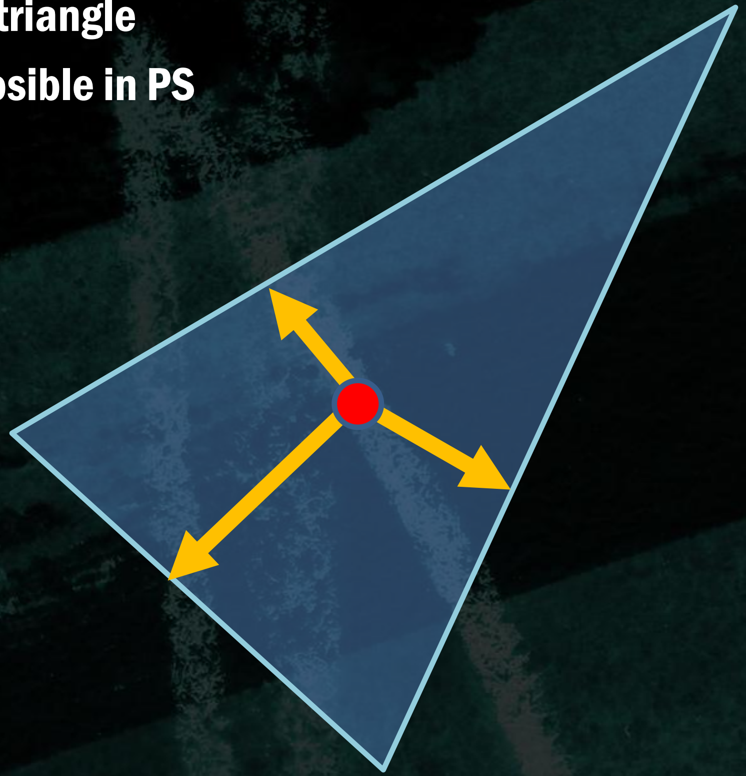
Interpolator : Packing

```
float4 Interpolate( float4 A, float4 B, float4 C, float3 barycentric )
{
   return A * barycentric.z + B * barycentric.x + C * barycentric.y;
}
```

```
   float3 barycentric;
   barycentric.xy = GetBarycentricCoordsPerspectiveCenter(); // Read Vi Vj from HW
   barycentric.z = 1 - barycentric.x - barycentric.y;

   uint rawA = ( GetVertexParameterP0( In.color_packed ) ); //Read Raw UINT Data from V00
   uint rawB = ( GetVertexParameterP1( In.color_packed ) ); //Read Raw UINT Data from V01
   uint rawC = ( GetVertexParameterP2( In.color_packed ) ); //Read Raw UINT Data from V10

   float4 decompressedA = UnpackColor( rawA ); //Unpack Byte from UINT and convert to float
   float4 decompressedB = UnpackColor( rawB ); //Unpack Byte from UINT and convert to float
   float4 decompressedC = UnpackColor( rawC ); //Unpack Byte from UINT and convert to float

   float4 Out;
   Out = Interpolate( decompressedA, decompressedB, decompressedC, barycentric );
```

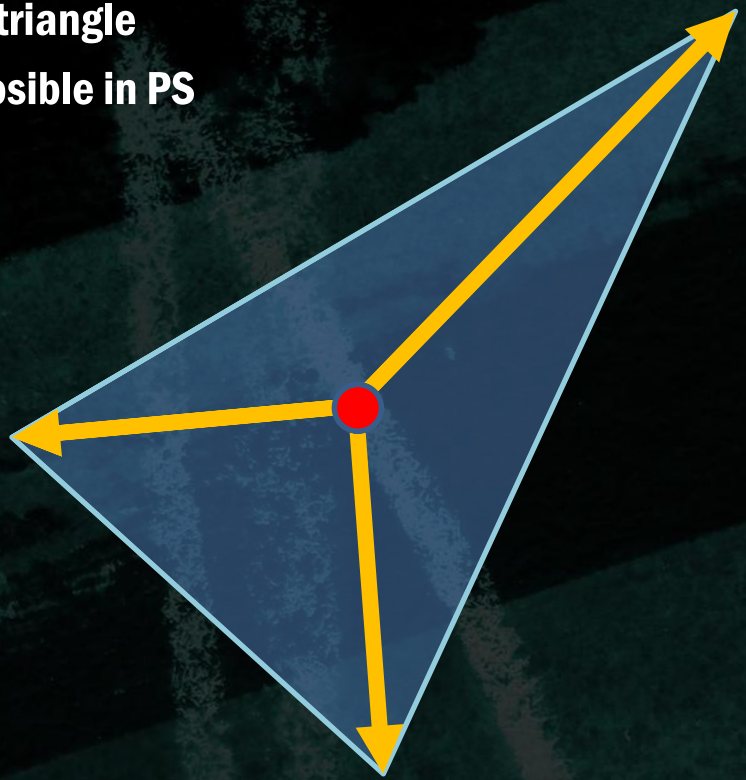
PS LDS access : Triangle Data

- PS can read Vertex Data directly from rasterized triangle
- Multiple algorithms previously reserved for GS posible in PS
  - Parallax Curvature estimation
  - (Closest) Distance to Edge
  - (Closest) Distance to Vertex
  - Spline Interpolated Normals / Curvature



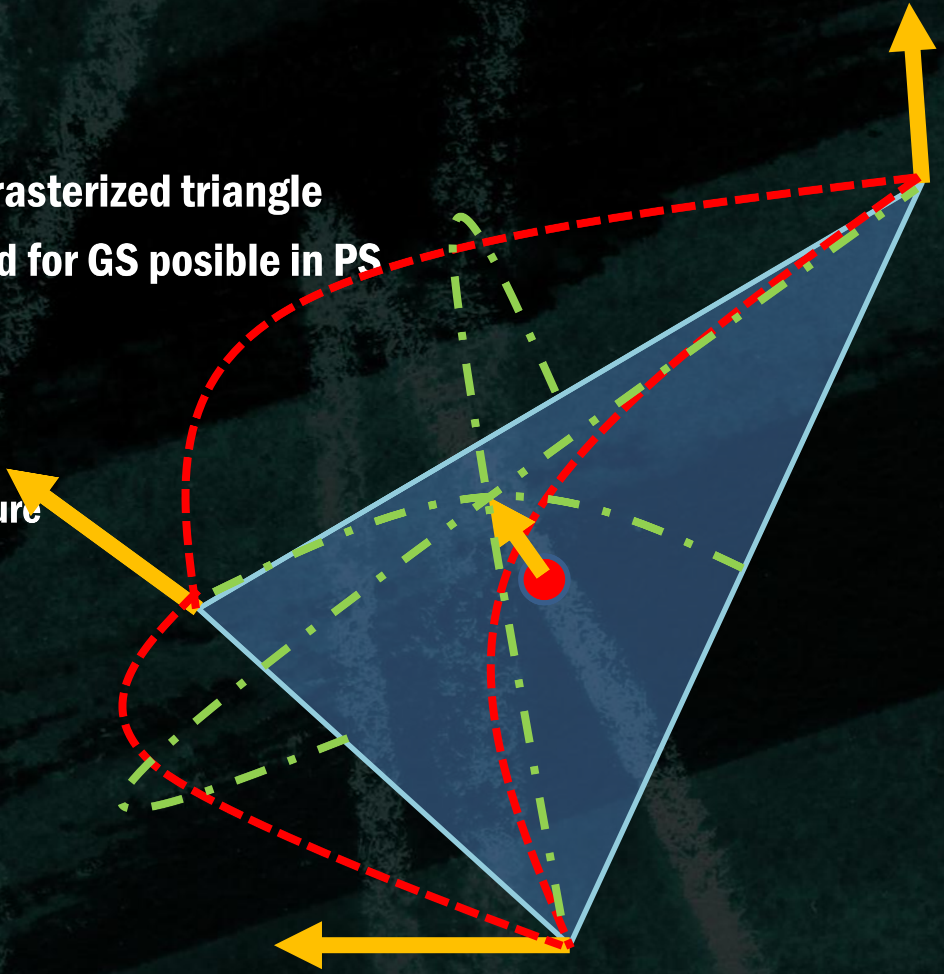
PS LDS access : Triangle Data

- PS can read Vertex Data directly from rasterized triangle
- Multiple algorithms previously reserved for GS posible in PS
  - Parallax Curvature estimation
  - (Closest) Distance to Edge
  - (Closest) Distance to Vertex
  - Spline Interpolated Normals / Curvature



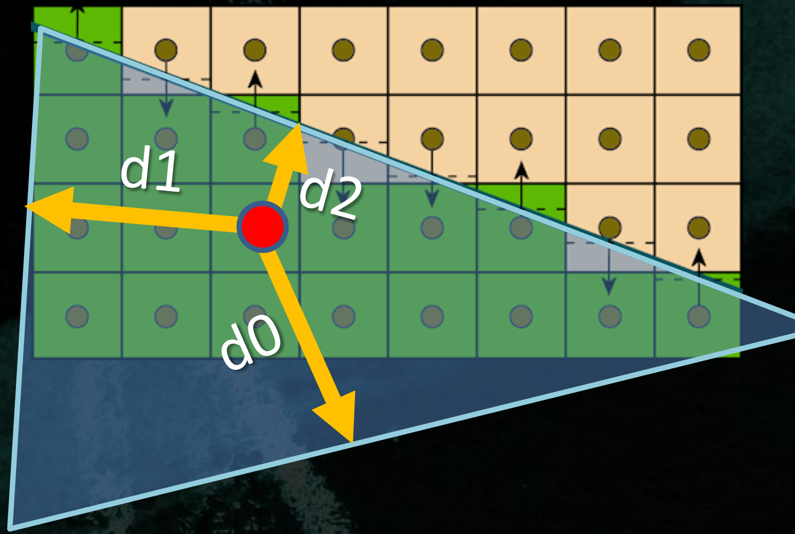
PS LDS access : Triangle Data

- PS can read Vertex Data directly from rasterized triangle
- Multiple algorithms previously reserved for GS posible in PS
  - Parallax Curvature estimation
  - (Closest) Distance to Edge
  - (Closest) Distance to Vertex
  - Spline Interpolated Normals / Curvature



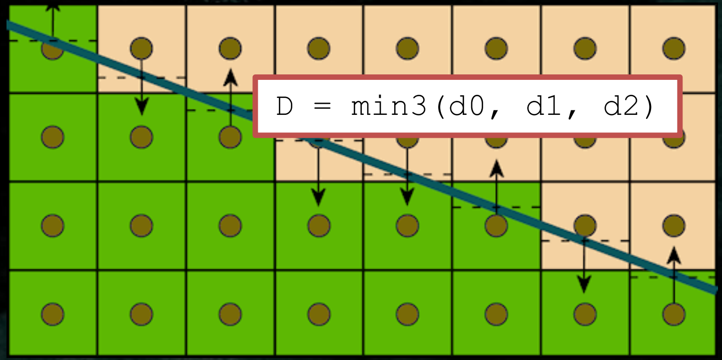
PS LDS access : Distance to Edge

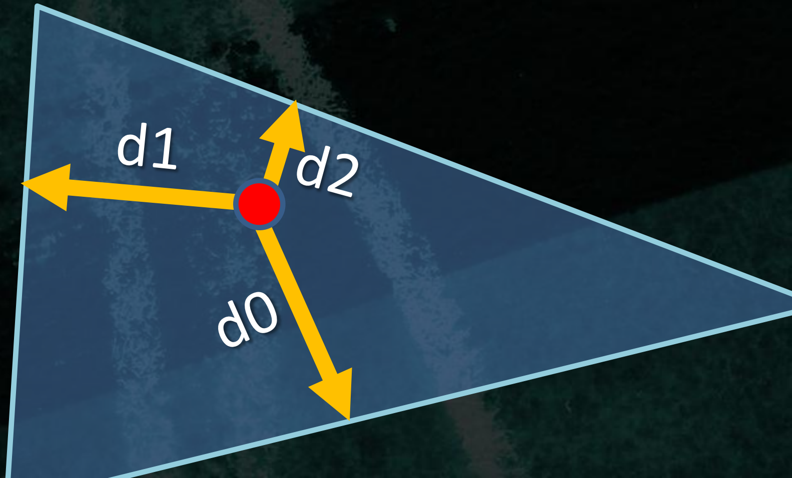
- **Example : Distance To Edge AA**
  - **Output distance to closest edge**
  - **Directly from PS bypassing GS**
  - **Used in multiple analytical AA methods**
    - **GBAA**
    - **DEAA**



PS LDS access : Distance to Edge

- **Example : Distance To Edge AA**
  - **Output distance to closest edge**
  - **Directly from PS bypassing GS**
  - **Used in multiple analytical AA methods**
    - **GBAA**
    - **DEAA**





PS LDS access : Distance to Edge

- Previously impractical due to expensive GS on every mesh
- Now totally viable option
  - Excellent performance
- Check HUMUS GBAA
  - Just move Geometry Shader part to Pixel Shader

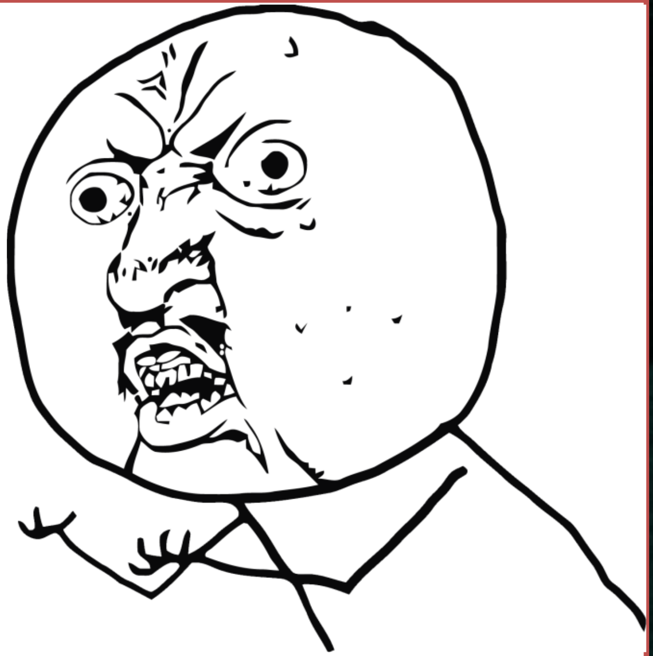
Transcendental Functions

- rcp(x), sqrt(x), rsqrt(x)
  - Most common transcendental functions in rendering
  - @Quarter Rate – 16 cycles each
  - Common in loops
    - Light iterators
    - SSAO
    - Multisampling
    - Raymarching
  - Used by macro
    - Length(x)
    - Normalize(x)

Transcendental Functions : Example

```
// SLOW code – some compilers are not aggressive enough to optimize macros
float3 vector;
float  vectorLength = length(vector);    // compiler best case :
                                          //expands to sqrt(dot(vector,         vector))
float3 normalVector = normalize(vector); // compiler best case : expands to
                                         // vector * rsqrt(dot(vector, vector))
```

```
// Timings : (FR - Full Rate cycle - 4 cycles):
v_mov_b32     v0, s2                // 1FR
v_mul_f32     v1, s2, v0            // 1FR
v_mov_b32     v2, s1                // 1FR
v_mac_f32     v1, s1, v2            // 1FR
v_mov_b32     v3, s0                // 1FR
v_mac_f32     v1, s0, v3            // 1FR
v_sqrt_f32    v1, v1                // 4FR
v_mul_f32 v0, s2, v0                // 1FR
v_mac_f32 v0, s1, v2                // 1FR
v_mac_f32 v0, s0, v3                // 1FR
v_rsq_f32     v0, v0                // 4FR
v_mov_b32     v2, #0x7f7fffff       // 1FR
v_mov_b32     s3, #0xff7fffff       // 1FR
v_med3_f32    v0, v0, s3, v2        // 1FR
v_mul_f32 v2, s0, v0                // 1FR
v_mul_f32 v3, s1, v0                // 1FR
v_mul_f32 v0, s2, v0                // 1FR
// Total not counting MOV           //18 FR
```



Transcendental Functions : Example

```
// help compiler by manually unrolling macros
// this is always a good practice
float  dotVector    = dot(inVector,inVector);
float  vectorLength = sqrt(dotVector);
float3 normalVector = inVector * rcp(vectorLength);
```

```
// Timings : (FR - Full Rate cycle - 4 cycles):
v_mov_b32     v0, s2        // 1 FR
v_mul_f32     v0, s2, v0    // 1 FR
v_mov_b32     v1, s1        // 1 FR
v_mac_f32     v0, s1, v1    // 1 FR
v_mov_b32     v1, s0        // 1 FR
v_mac_f32     v0, s0, v1    // 1 FR
v_rsq_f32     v1, v0        // 4 FR
v_sqrt_f32    v0, v0        // 4 FR
v_mul_f32     v2, s0, v1    // 1 FR
v_mul_f32     v3, s1, v1    // 1 FR
v_mul_f32     v1, s2, v1    // 1 FR
// Total not counting MOV   //14 FR
```

Transcendental Functions : Example

```
// We can do much better for FR count and pipelining by exploiting:
// sqrt(x) = rsqrt(x) * x
// rcp(x)  = rsqrt(x) * rsqrt(x) // only for positive X
float  dotVector         = dot(inVector,inVector);
float  rcpVectorLength   = rsqrt(dotVector);
float  vectorLength      = rcpVectorlength * dotVector;
float3 normalVector      = inVector * rcpVectorLength;
```

```
// Which results in:
v_mov_b32     v0, s2      // 1 FR
v_mul_f32     v0, s2, v0  // 1 FR
v_mov_b32     v1, s1      // 1 FR
v_mac_f32     v0, s1, v1  // 1 FR
v_mov_b32     v1, s0      // 1 FR
v_mac_f32     v0, s0, v1  // 1 FR
v_rsq_f32     v1, v0      // 4 FR
v_mul_f32     v0, v0, v1  // 1 FR
v_mul_f32     v2, s0, v1  // 1 FR
v_mul_f32     v3, s1, v1  // 1 FR
v_mul_f32     v1, s2, v1  // 1 FR
// Total not counting MOV  //11 FR
```

Approximated Transcendental Functions

- Transcendental Functions in HW provide ~1 ULP of precision
- We do not always need that much
  - Especially for F16, F11, UNorm8 data

- Can we do a better job than HW @ Quarter Rate?

Special ALU Ops : Integer Math

- **General Purpose Registers**
  - **Integer Math**
  - **No reinterpretation cost**
- **Integer support**
  - **Allows integer based floating point math**

```
// asint() / asfloat() works as reinterpret_cast
// is free - just hints the compiler to treat the data using different instruction set

#define asint(_x)   *reinterpret_cast<int*>(&_x);
#define asfloat(_x) *reinterpret_cast<float*>(&_x);
```

0x5f3759df WTF?

- **Fast Inverse Square Root**
  - **Implemented at SGI using integer math**
  - **Famous due to Quake 3 source code**

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // Float to Int reinterpret
        i  = 0x5f3759df - ( i >> 1 );               // WTF?
        y  = * ( float * ) &i;                      // Int to Float reinterpret
        y  = y * ( threehalfs - ( x2 * y * y ) );   // Newton Raphson 1st iteration

        return y;
}
```

0x5f3759df WTF?

- **Fast Inverse Square Root**
  - **Works due to Floating Point Number Binary representation**
- **Care about speed**
  - **Remove Newton-Raphson iteration**
- **Should be fast on GCN?**
  - **2x faster than rsqrt()**

```
int x = asint(inX);
x = 0x5f3759df - (x >> 1);
return asfloat(x);
////////////////////////////////
 v_ashr_i32    v0, v0, 1
 v_sub_i32     v0, # 0x5f3759df, v0
```

More Magic!

- **Using original idea derive**
  - $x^n \approx qpow(x,n) = K + n(asInt(x) - K)$, $n: [-1, 1]$
  - **K is a constant**

- $E(x,n) = |x^n - qpow(x,n)|$
- **We search for K to minimize E(x, n) over (x,n) domain**

- $E(K) = \sum_x E(x,n)$, $n = const$ $-$ *has stationary point*
- **asInt(x) is ‚close' to log function**
  - **E(K) has global minima for given x domain and n**

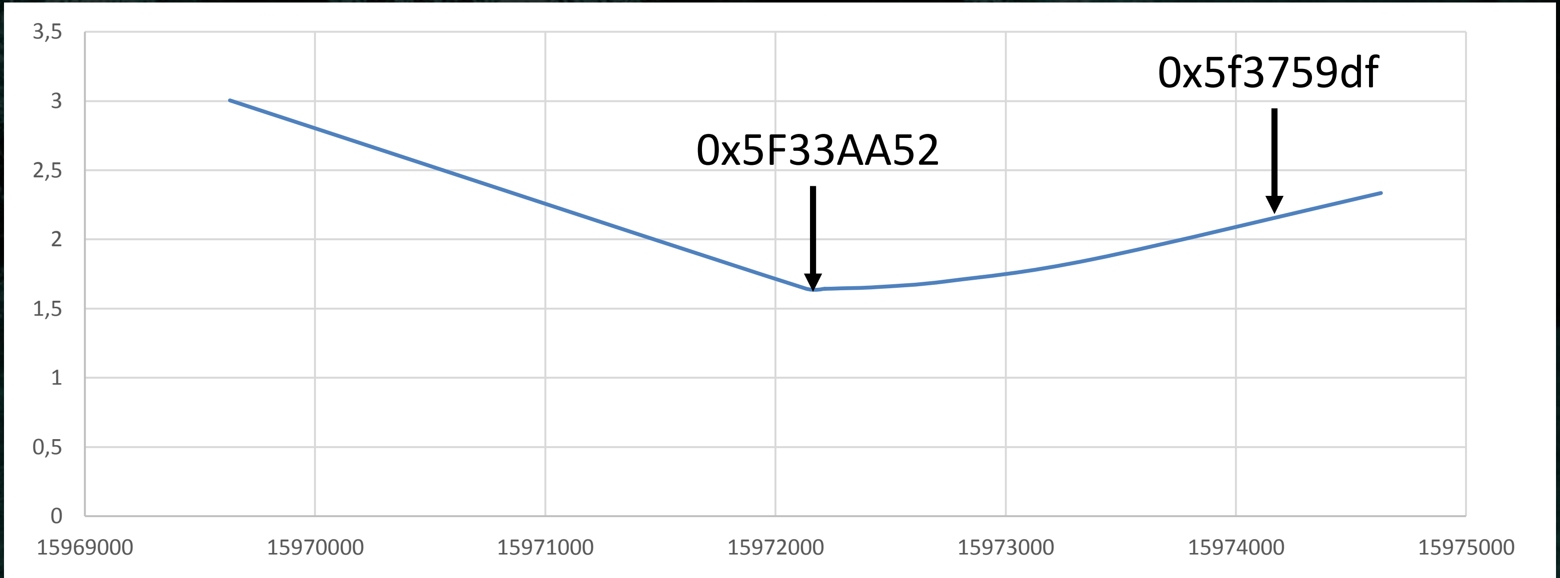
More Magic!

- Using gradient binary search
  - Find K best for your
    - n
    - x - domain
  - Can find reasonable K for all
- Recommended specialization to minimize error function
  - Find best K for sqrt(), rsqrt(), rcp()
  - Limit domain
    - i.e. For distance calculation in Camera Space – cap x to Far Plane

Let's beat 0x5f3759df RSQRT()

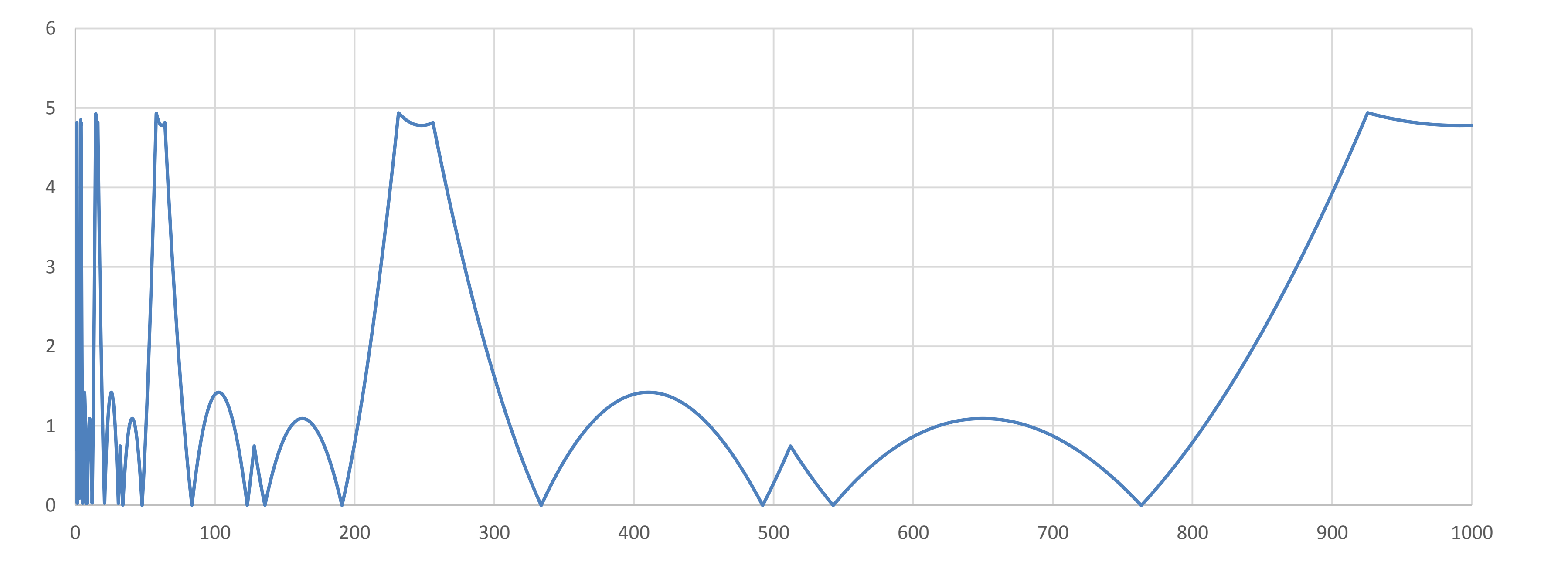
- 0x5f3759df – found as universal K for rsqrt()
- Our domain is limited x (0, 1000)

- RMSE(K) in % : x(0, 1000), n = -1/2



Let's beat 0x5f3759df RSQRT()

- **E(0x5F33AA52) , x(0,1000) , n = -1/2**



Fast Shader Lib

```
// 2 Full Rate
float rcpSqrtIEEEIntApproximation(float inX, const int inRcpSqrtConst)
{
	int x = asint(inX);
	x = inRcpSqrtConst - (x >> 1);
	return asfloat(x);
}
// 2 Full Rate
float sqrtIEEEIntApproximation(float inX, const int inSqrtConst)
{
	int x = asint(inX);
	x = inSqrtConst + (x >> 1);
	return asfloat(x);
}
// 1 Full Rate
float rcpIEEEIntApproximation(float inX, const int inRcpConst)
{
	int x = asint(inX);
	x = inRcpConst - x;
	return asfloat(x);
}
```

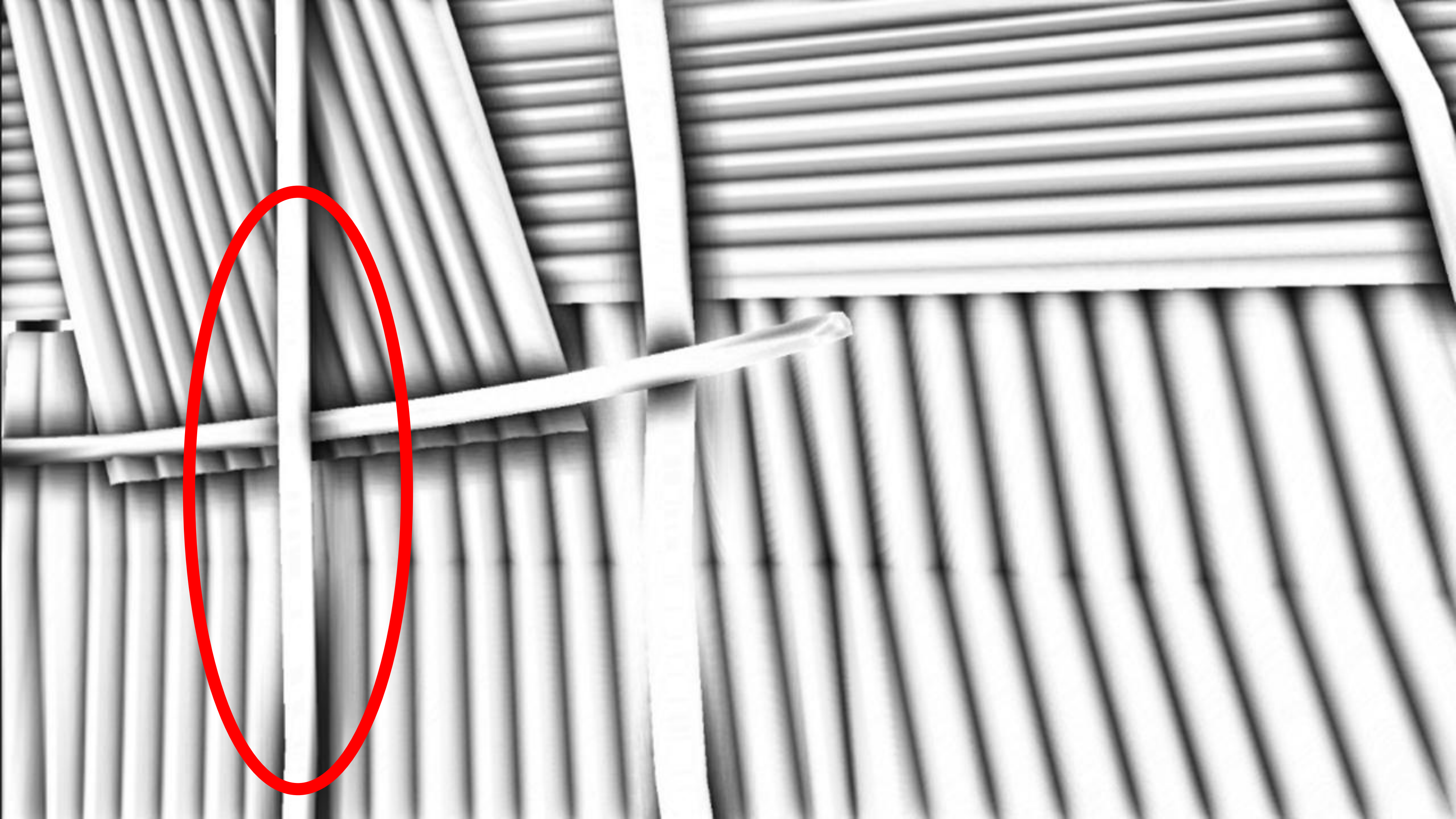
Use Case : Example K's

- rsqrt()
  - 0x5F341A43 RME:1.72% (0.0, 1.0)
  - 0x 5F33E79F RME:1.62% (0.0, 1000.0)
- sqrt()
  - 0x1FBD1DF5 RME:1.42% (0.0, 1.0)
  - 0x1FBD22DF RME:1.44% (0.0, 1000.0)
- rcp()
  - 0x7EEF370B RME:2.92% (0.0, 1.0)
  - 0x7EF3210C RME:3.20% (0.0, 1000.0)

Use Case : SSAO / Bilateral Filter

- SSAO
  - Distance()   sqrt()
  - Normalize()   rsqrt()
- Bilateral Filter
  - Divide()   rcp()
  - Normalize()   rsqrt()

- All switched to Fast Shader Lib
  - 13% total time improvement on Consoles
  - No visible difference



Be a Creative Code Ninja!

- GPU so much closer to CPU
- Use your SPU / CPU Ninja Skillz!
- Old things might surprise you
- Trade ALU for BW

- Tip of the iceberg
  - Scheduling
  - Async compute
  - Latency Hiding
  - Caching
  - Tons of things we don’t know yet
  - Fun ahead!

Q&A

- More Tips & Tricks :
- MICHALDROBOT.COM

- Catch me on TWITTER
- @MichalDrobot

- Shoot me email
- HELLO@DROBOT.ORG

References

- GCN
  - „Low-level Shader Optimization for Next-Gen and DX11" – Emil Persson
  - „The AMD GCN Architecture: A Crash Course" - Layla Mah
  - „GCN – Two ways of latency hiding and wave occupancy" – Bart Wronski
  - „Compute Shader Optimizations for AMD GPUs: Parallel Reduction" – Wolfgang Engel
  - GCN Performance Tweets
- Inverse Sqrt
  - „Fast inverse Square Root" - Chris Lomont
  - "The Mathematics Behind the Fast Inverse Square Root Function Code" – Charles McEniry
  - Quake 3 Source Code - github.com/id-Software/Quake-III-Arena

Thanks!

- Ubisoft 3D Team(s)
- Especially:
  - Bart Wronski
  - Jeremy Moore
  - Steve McAuley
  - Stephen Hill

- AMD Developer Relation Team
- Especially:
  - Layla Mah
  - Chris Brennan

Bonus Slides

IEEE Performance Mode

- Disable IEEE compliance (-fastmath) to enable VOP3
  - Also called IEEE strict
  - Compiler will NOT handle
    - Denormals
    - QNaNs
    - Div 0
    - Other unsafe cases
  - Will use approximate Transcendental Functions
    - Without cleanup or accuracy OPs
    - Precision varies but guaranteed to be ~1 ULP (IEEE requires 0.5 ULP)

IEEE Strict vs Non-Strict : X / Y

```
float r = inV.x / inV.y;
```

```
// Without IEEE strict
// x - v1 y - v2
v_rcp_f32      v0, v1          // Unsafe rcp()might produce NaN
v_mul_f32      v0, v2, v0
```

```
// IEEE strict safe -fastmath
// x - v1 y - v2
v_rcp_f32      v0, v1          // Unsafe rcp()might produce NaN
v_mov_b32      v1, #0x7f7fffff // MAX_FLT
s_mov_b32      s1, #0xff7fffff // MIN_FLT
v_med3_f32     v0, v0, s1, v1  // safe clamping to clean NaNs
v_mul_legacy_f32 v0, v2, v0
```

IEEE Strict vs Non-Strict : X / Y

```
float r = inV.x / inV.y;
```

```
// IEEE strict safe accurate flush denormals
// x - v1 y - v2
v_rcp_f32            v0, v1
v_mul_f32            v0, v0, v2
v_div_fixup_f32      v3, v0, v1, v2 // Fix -/+ INF NaN QNaN
```

IEEE Strict vs Non-Strict : X / Y

```
float r = inV.x / inV.y;
```

```
// IEEE strict safe accurate support denormals
// depending on rounding modes and denormal output
// compiler can add:
v_rcp_f32
v_mul_f32
v_div_scale_f32
nop
nop
nop
nop
v_div_fmas_f32
```