ATI Radeon™ HD 2000 programming guide

Emil Persson
AMD Graphics Products Group
emil.persson@amd.com

Introduction

The ATI Radeon™ HD 2000 series is a major upgrade to AMD's graphics product line. It is based on a new architecture and was built with full hardware support for the Direct3D® 10 API. This guide goes through the performance characteristics of this new chip and aims to help developers achieve the best possible performance.

What's new?

The most significant difference between the ATI Radeon HD 2000 and earlier hardware generations is that it adds full support for Microsoft's Direct3D 10 API. The ATI Radeon HD 2000 is built on a new architecture and has many performance enhancements under the hood that changes its performance characteristics compared to earlier generations. One important such enhancement is that the shader core now is fully unified, meaning that the vertex shader, geometry shader and the pixel shader all share the same shader resources. The hardware automatically schedules work onto the shader pipelines to balance the workload. This means that if the workload on the pixel shader is light, more computation power will be shifted over to the vertex shader and geometry shader and vice versa.

Another important change compared to earlier generations is that the shader core now is scalar based whereas previous generations were vector based. The X1000 series could execute one 3-component vector and one scalar instruction in parallel in the pixel pipeline meaning that scalar-heavy shaders would be able to process at most two scalar instructions per cycle. In the vertex shader it was similar, with the difference that it was a 4-component vector plus a scalar. The ATI Radeon HD 2000 series on the other hand has 5 scalar units where the fifth unit has extra functionality over the other four.

Other improvements include a wider bus, enhanced video capabilities, on-chip Crossfire support and a wide range of performance enhancements across the board. We will discuss some of the important changes and their implications for developers in this document.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

Shaders

Generic shader optimization

With a unified architecture many shader optimizations are fundamentally the same between the vertex, geometry and pixel shader. Since the same operations in the different shaders run through the same hardware you can expect similar performance in all units. This means that for instance vertex texture fetch should run at about the same speed as you expect it to run in the pixel shader.

Unified architecture

Since the shader core is unified all shaders consume shader processing power from the shared pool of processing resources. This means that optimizing the code that's not the main bottleneck may still result in a performance increase. On earlier hardware generations one would for instance not care too much about the vertex shader in a very pixel shader limited case since the shaders ran on separate hardware units. So if the pixel shader was saturated, one could view the vertex processing as being for free since adding more work to the vertex shader did not change the performance. With a unified shader architecture this is not true anymore. Now it's the total amount of work that matters instead of the amount of work in the heaviest loaded unit. So an unoptimized vertex shader will drain computational resources to some degree from the pixel shader even in a pixel shader bound case. This makes it important to look at all shaders and not just the most dominant one.

Dynamic branching

The ATI Radeon HD 2000 hardware has excellent dynamic branching performance for all three shader stages. Using dynamic branching you can reduce the workload in e.g. a pixel shader by skipping past instructions that don't need to be executed. In a lighting situation, if the pixel is in shadow you don't need to compute the whole lighting equation but may return zero or the ambient value immediately. Same thing if the pixel is beyond the light radius. It can be beneficial to rearrange your code so that you can do some large scale culling early in the shader. For instance attenuation is usually cheap to compute, so it makes sense to do that first in the shader. Depending on how your shader looks you may sometimes see better performance if you stick to a small set of branches instead of several branches shortly after each other. Thus it may be faster to multiply the attenuation with the shadow factor and check that value against zero, rather than checking attenuation first and then shadow in a separate branch. This varies a lot with the situation, so it's recommended that you try both approaches and see which one comes out faster in your application.

One thing to keep in mind though is that branches need to be coherent to achieve top performance. If pixels within the same thread take different branches the hardware will have to execute both sides of the branch and just select the results for each pixel. So for branches that generally are not coherent you will probably see a performance loss compared to code without branching. For the ATI Radeon HD 2000

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

you need a coherency of at least 64 pixels or vertices. Anything that varies in smaller units than that in general should not use dynamic branching. For example you may have the following code:

```
float diffuse = dot(lightVec, normal);
if (diffuse > 0.0)
{
    // Compute lighting ...
}
```

If the normal is an interpolated normal across the surface this branch is fine. But if the normal comes out of a high-frequency normal map this code may result in a performance loss. This is because normals from a high-requency normal map can typically vary a lot from pixel to pixel. As a result, in most cases the hardware will not be able to skip past the code within the if-statement, so there is no performance gain to be had, but you incur a small performance hit from doing the actual branch test and possible additional register pressure.

Scalar ALUs

The shader core has 5 scalar units and the most common math operations can execute on all these units. However, certain less frequently used operations only run on the fifth unit, which is also known as the transcendental unit and is equipped with more capabilities. Operations that only run on the fifth unit is integer multiplication and division, bit shifts, reciprocal, division, sqrt, rsqrt, log, exp, pow, sin, cos and type conversion. Moderate use of these is OK, but if a shader is dominated by these operations, the other scalar units will go idle resulting in much lower throughput.

To maximize the performance you should prefer using instructions that can run on all five units. In some cases integer operations can be changed for floating point, which may allow it to run on all units. Note that integer additions and subtractions can run on all units though, as well as logical operations such as and/or/xor/not. If the data you're computing is going to be converted, you may be able to defer some operations to after the conversion to allow the use of better instructions. One common case is extracting packed data. Assume you have an RGBA color packed in a uint. A standard extraction approach might look something like this:

```
uint r = (color      ) & 0xFF;
uint g = (color >>  8) & 0xFF;
uint b = (color >> 16) & 0xFF;
uint a = (color >> 32) & 0xFF;
return float4(r, g, b, a) * (1.0 / 255.0);
```

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
Smarter Choice

This compiles down to 8 instruction slots. We have three bit-shifts and four conversions that are responsible for this. We can't easily get rid of the conversions, but the bit-shifts can be changed for floating multiplications after the conversion.

```
uint r = color & 0x000000FF;
uint g = color & 0x0000FF00;
uint b = color & 0x00FF0000;
uint a = color & 0xFF000000;
return float4(r, g, b, a) * (1.0 / (255 * float4(1, 256, 65536, 16777216)));
```

This code compiles down to 6 instructions slots, while also leaving more scalar slots open for the compiler to fit other code into. In particular, we now only have 4 instructions (for the type conversions) that need to execute on the transcendental unit compared to 7 for the original code.

Write parallel code

For previous generations the recommendation was to vectorize the code. For this generation the recommendation is to write parallel code. Vectorized code is still good, because it's also parallel; however, you may write code that's not vectorized but still very parallel. Forcing operations into vectors may turn out counterproductive on a scalar architecture though. On the other hand, there's no need to break code that's naturally vectorized into scalars. In general it's recommended that you keep operations in their natural domain.

It's important to not assume that because there are 5 independent scalar units you will always be able to crunch through the math at 5 scalar operations at a time. Depending on what the shader does you may at worst not be able to execute more than one scalar in parallel. Consider this simple code:

```
float x = a + b + c;
```

Even though this is just two scalar operations it will still require two instruction slots because the first addition has to be completed before the second can take place. Thus the code is sequential and in practice is executed as follows:

```
float t = a + b;
float x = t + c
```

One thing to consider is that HLSL evaluates expressions left to right, just like C/C++. For some expressions this could matter. Take for example the following code:

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
Smarter Choice

```
float x = a + b + c + d;
```

Left to right evaluation makes this code sequential and equivalent to this:

```
float t = a + b;
float u = t + c;
float x = u + d;
```

We can introduce explicit parallelism to the code simply by using parentheses.

```
float x = (a + b) + (c + d);
```

Now the a + b and c + d additions can be performed in parallel on different scalar units, which reduces this expression from 3 slots to 2. Shader optimizers may be able to do this automatically for you in some cases, but doing this explicitly improves the shader compiler's ability to issue parallel instructions. To see what code is actually generated for a shader you can use the AMD GPU ShaderAnalyzer tool.

When mixing scalars and vectors the order of operations can matter even more. It is recommended that you do as much of the scalar work first before expanding the math to vectors. Consider this for example:

```
float4 a;
float b, c;
...
float4 x = a * b * c;
```

This code would be equivalent to this:

```
float4 t = a * b;
float4 x = t * c;
```

This amounts to 8 scalar multiplications. This can be improved by adding parentheses:

```
float4 x = a * (b * c);
```

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

AMD Smarter Choice

That is equivalent to this:

```
float t = b * c;
float4 x = a * t;
```

This amounts to only 5 scalar multiplications. Optimizations like this are not always obvious, but the performance improvement could be very substantial. On previous generation hardware you would probably not see nearly as big improvement, even though it could help there too in some cases. Always keep in mind what types your variables are and try to keep as many operations as possible as scalars before expanding to vectors. Out of old habit it is easy to think of vector operations as not being any more expensive than scalars just because they are a single instruction from the D3D runtime point of view, but that is not true. In fact, many “single instructions” in D3D10 are actually expanded to multiple hardware instructions under the hood, like for instance a vector division could add up to five instruction slots, while in other cases up to five instructions may be packed into a single instruction slot.

Use the right data type

In relation to the above discussion it’s also important to remember to use the right data type for your data. One might habitually make most or all variables float4, especially if you transitioned from programming shaders in assembly, which will result in many more instructions used than necessary. Also, if you fetch data from textures you don’t necessarily have to store that in float4 just because the function returns a float4. If you know what texture formats you will use for a particular texture, for instance if you use only luminance or you only care about RGB, then there is no reason to store the superfluous components.

Avoid mixing types

Type conversions are only available in the transcendental unit. This means that if you do a lot of type conversions performance will suffer. Small mistakes can sometime have a large performance impact. Consider this code:

```
int4 main(int4 a: TEXCOORD) : SV_Target
{
      return a + 1;
}
```

The above code consumes a single instruction slot. Now imagine that the developer did this mistake:

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

```
int4 main(int4 a: TEXCOORD) : SV_Target
{
        return a + 1.0;
}
```

The code now takes 8 instruction slots. This is because 1.0 is a float, so the variable "a" will be converted to float4 (which requires 4 conversions), then the addition will be done in floating point and finally the result converted back to the return type int4 (another 4 conversions).

Keep shader length reasonable

While D3D10 has no upper limit on how long shaders can be, unlike D3D9, it does not mean it is a good idea to make extremely long shaders. The shader cache like any other cache has limited storage. So very long shaders could spill out of the cache and reduce performance. If the shader is in the range of several thousand instructions the throughput will go down as a result.

Indexed constants

Depending on the access pattern it is important to consider how you are bringing data into the shader. Keep in mind that constant buffer are typically accessed directly and thus have full coherency across the thread. Also, when accessed with indices it is typically fairly coherent. Thus constant buffer access has been optimized for coherent accesses. Textures on the other hand are more prone to scattered accesses and thus are more capable of coping with such an access pattern. If a certain set of data is accessed in a very random fashion it may be faster to use texture accesses than indexed constants.

Indexed temporaries

Shader Model 4.0 adds support for indexed temporaries which can be quite useful for certain tasks. However, don't make a habit of using them where not necessary. Regular direct temporary access is preferable is most cases. One reason is that indexed temporaries are hard to optimize. The shader optimizer may not be able to identify optimizations across indexed accesses that could otherwise have been detected. Furthermore, indexed temporaries tend to increase register pressure a lot. An ordinary shader that contains for instance a few dozen variables will seldom consume a few dozen temporaries in the end but is likely to be optimized down to a handful depending on what the shader does. This is because the shader optimizer can easily track all variables and reuse registers. This is typically not possible for indexed temporaries, thus the register pressure of the shader may increase dramatically. This could be detrimental to performance as it reduces the hardware's ability to hide latencies. To avoid this problem the shader optimizer may decide to use memory instead of temporaries which of course has performance implications of its own.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

Smarter Choice

Also keep in mind that the HLSL compiler interprets code much more literally than it used to do for previous shader models. So just because a loop can easily be unrolled it doesn't mean it will be. In fact it likely won't be unless it for some reason can't be compiled as a loop. As a result, if you access an array in the shader that you expected to be unrolled you may end up with a loop that contains indexed temporaries. This could slow down your code significantly. If this happens you can force the HLSL compiler to unroll the loop with the [unroll] tag.

Vertex shader

Vertex texture fetch

Since the ATI Radeon HD 2000 is a unified architecture the same hardware used for texturing in the pixel shader is used in the vertex shader, so you will see the same performance you would expect with the same fetches from the pixel shader. This means that using vertex texture fetch is not just fast, but in many cases highly recommended. If you are vertex fetch limited it may be faster to distribute the data between a vertex buffer and a texture. In D3D10 you could use SV_VertexID to derive the texture coordinate so no extra data is needed for the texture lookup. The reason this is potentially faster is that there is different hardware for vertex fetch and texture fetch. If you are not bound by bandwidth for these fetches you could potentially double your input rate.

Another reason for considering using vertex texture fetch is that you have a wider choice of formats, including compressed formats. In some cases you may also be able to take advantage of texture filtering or reduce data storage by using a smaller resolution. Using vertex texture fetch you have more options for making a more compact representation of your vertex storage.

Geometry shader

The geometry shader is a new pipeline stage available in Direct3D 10 that offers tremendous amount of flexibility for primitive processing. However, this flexibility comes with certain caveats to keep in mind as the performance characteristics of this shader are quite different from the other two shaders. In the traditional model where a vertex shader writes outputs to the pixel shader it is written to on-chip buffers so that there is no memory bandwidth consumed for communication between stages. However, the geometry shader breaks away from this simple model with small and known inputs and outputs. A geometry shader could produce any number of primitives within the declared maximum vertex count. For this reason the geometry shader often spills to memory. As a result, writing out the outputs is often the bottleneck. There are certain exceptions to this rule. When the geometry shader does 1:1 input/output, i.e. does no amplification or reduction, the shader can run with similar performance characteristics as a regular vertex shader, and similarly since the geometry shader is expected to be commonly used for replacing point sprites (by expanding point primitives to a triangle strip with two

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
Smarter Choice

triangles) there is also special hardware for taking care of the 1:4 case. These two cases are subject to certain restrictions:

- The shader uses minimal or no flow control.
- The shader does not use SV_PrimitiveID, SV_ClipDistance, SV_CullDistance, SV_RenderTargetArrayIndex, or SV_ViewportArrayIndex.
- The shader does not use resources (textures, buffers, or constant buffers).
- The shader does not use primitives with adjacency.
- The shader is symmetrical. That is, for each input vertex, the shader applies the exact same sequence of instructions to produce the output vertex.

Some of these restrictions may be relaxed in future drivers.

Max vertex count

A geometry shader needs to declare its maximum vertex count. This sets the upper bound on how many vertices the shader outputs at most. This value is a hard limit and not primarily an optimization hint. Any vertex the shader might output beyond the declared maximum will be discarded. Therefore it is important that this value is not set too low. If you are experiencing missing geometry, make sure you have declared a sufficiently large maximum vertex count. The good news is that the ATI Radeon HD 2000 series is largely insensitive to the declared maximum vertex count, so for cases where the actual maximum would be hard to know on compile time you can set it to a safe upper bound without worrying about performance impact.

Keep data small

When outputting data from the geometry shader to the pixel shader it is important to keep each vertex small. The smaller the vertex, the more vertices you can output per pass, which means you can do more work per pass. Alternatively it means you don't use as much bandwidth. In either case, better performance can be had this way. In many cases the number of interpolators passed to the pixel shader can be reduced. A common case would be if a pixel shader needs both a light vector and a view vector. Conventional wisdom suggests that these should be computed in the vertex shader and passed to the pixel shader to reduce the amount of work in the pixel shader since they are linear and thus can be interpolated. This wisdom still holds true if you're using a vertex and pixel shader only. So you would probably do something like this in the vertex shader:

```
Out.position = mul(mvp, In.vertex);
Out.lightVec = lightPos - In.vertex.xyz;
Out.viewVec = camPos - In.vertex.xyz;
```

If you are using a geometry shader you are likely bound by the output though, which means you will usually benefit from trading a couple of instructions in the pixel shader for less data written from the

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

geometry shader. For the case with a light vector and a view vector it is better to only pass the position and then compute both vectors in the pixel shader. So the geometry shader would do this:

```
Out.position = mul(mvp, In.vertex);
Out.vertex = In.vertex.xyz;
```

And then the pixel shader would do this:

```
float3 lightVec = lightPos - In.vertex;
float3 viewVec = camPos - In.vertex;
```

Depending on the balance of work between geometry shader and pixel shader it may in some cases be beneficial to take this a step further. The position could be computed in the pixel shader from the SV_Position register by back-projecting it from screen space to world space. This is possible in D3D10 since the SV_Position defines z too (and w as well, but we don't need it), unlike the VPOS register in D3D9 which only defines x and y. The resulting pixel shader would look something like this:

```
float4 wp = mul(screenToWorld, float4(In.sv_position.xyz, 1));
float3 worldPos = wp.xyz / wp.w;

float3 lightVec = lightPos - worldPos;
float3 viewVec = camPos - worldPos;
```

While it is often the output that is the bottleneck the input bandwidth is also often important too depending on the amount of amplification. With relatively small amount of amplification, and especially in decimation cases, the input could be a significant bottleneck. It is important to keep the input vertices as small as possible. It is usually beneficial to trade a few extra instructions in the vertex and/or geometry shader to pack data. On the output side one problem is that interpolators are interpolated as floats, so packing data in the geometry shader and extracting it in the pixel shader is usually not possible unless it is static across a primitive. Between the vertex shader and the geometry shader we don't have that problem though, so packing data is practical and advantageous. For instance if you are passing a texture coordinate you may consider storing that in a single uint with 16 bits per component instead of using two floats. For a normal you can often pack that into a single uint as well, with 11, 11 and 10 bits for each channel. Best is if you store it this way in the vertex buffer already which also saves input bandwidth for the vertex shader, saves memory and takes the encoding cost out of the vertex shader.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

If you have data that is being processed in the geometry shader that is just passed through from the vertex shader with little or no processing you may consider placing that data elsewhere, such as in a texture or constant buffer, and access that in the geometry shader directly instead.

Merge vectors

It is also often beneficial to merge data together into single vectors. While this doesn't change the data size per se it can reduce the number of reads and writes the hardware issues to get the data in and out of memory. This is true on both input and output. For instance if you are passing two texture coordinates, instead of two separate float2 vectors you can pass the first in .xy and the second in .zw of a float4 vector. Or if you are passing a texture coordinate and a normal, the faster option is likely to pack the texture coordinate into a uint and place that in .w of the normal to make it a single vector. Since there are no mixed type vectors in HLSL you would have to use asfloat(texCoord) to place the uint into the float .w and then use asuint(normal.w) when you unpack it. This would naturally not work on output due to floating point interpolation in the pixel shader, but is a reasonable thing to do on input to the geometry shader.

Use frustum and back-face culling

The geometry shader can be used to optimize rendering by doing more work in a single pass. For instance you can reduce the number of draw calls by up to a factor of six for render-to-cubemap cases by transforming each input triangle with the MVP matrix for each face and outputting to each face. To further optimize this you should implement frustum culling and back-face culling in the geometry shader to avoid outputting more primitives than necessary. Keep in mind that most primitives would only need to be output to one cubemap face. Theoretically a triangle could be in as many as 5 faces, but in a reasonably detailed scene the number of output faces per triangle is just slightly above one. Also, on average about half of the triangles would be back-facing. This means you can get performance several times higher than a naïve implementation that is writing out all primitives to all faces. A conservative culling can be implemented in relatively few instructions. Here is an example implementation:

```
[maxvertexcount(18)]
void gsCube(triangle GsIn In[3], inout TriangleStream<PsIn> Stream)
{
    PsIn Out;

    [unroll]
    for (int k = 0; k < 6; k++)
    {
        Out.face = k;

        float4 pos[3];
        pos[0] = mul(mvpArray[k], In[0].pos);
        pos[1] = mul(mvpArray[k], In[1].pos);
```

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

AMD Smarter Choice

```
        pos[2] = mul(mvpArray[k], In[2].pos);

        // Use frustum culling to improve performance
        float4 t0 = saturate(pos[0].xyxy * float4(-1, -1, 1, 1) - pos[0].w);
        float4 t1 = saturate(pos[1].xyxy * float4(-1, -1, 1, 1) - pos[1].w);
        float4 t2 = saturate(pos[2].xyxy * float4(-1, -1, 1, 1) - pos[2].w);
        float4 t = t0 * t1 * t2;

        [branch]
        if (!any(t))
        {
            // Use back-face culling to improve performance
            float2 d0 = pos[1].xy * pos[0].w - pos[0].xy * pos[1].w;
            float2 d1 = pos[2].xy * pos[0].w - pos[0].xy * pos[2].w;
            float w = min(min(pos[0].w, pos[1].w), pos[2].w);

            [branch]
            if (d1.x * d0.y > d0.x * d1.y || w <= 0.0)
            {
                [unroll]
                for (int i = 0; i < 3; i++)
                {
                    Out.pos = pos[i];
                    // Fill output structure here ...
                    Stream.Append(Out);
                }

                Stream.RestartStrip();
            }
        }
    }
}
```

The above code has been highly tuned to run efficiently and not require additional inputs. For each face, all input vertices are first transformed by the appropriate MVP matrix. This would have to be done anyway. With the positions in clip space the X and Y positions are compared to W coordinate. The frustum is the area where $-W < X < W$, and similarly for Y. In depth direction it would be $0 < Z < W$ (or $-W < Z < W$ in OpenGL). We don't check for Z in this code though. Depending on your situation you may want to add a check for Z as well. In our tests this did not improve performance, but your mileage may vary. Each component in t0, t1 and t2 is a test for one screen edge for each vertex. By multiplying these together we end up with four values saying whether the triangle was fully behind that edge plane. If we are not fully behind any plane, we move on to the back-face test. Note that this is just a conservative check. A triangle may not be fully behind any individual plane but still be outside the frustum. However, this test will catch the vast majority of the triangles with few instructions. Also note that even though we don't check in Z direction most triangles behind the near clipping plane will be

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
Smarter Choice

culled by either of the X and Y planes anyway as they intersect at W = 0 and leave only a small pyramid behind the near plane unaccounted for.

The back-face culling code is also a conservative check. It will work as long as W > 0 for all vertices. Most triangles where this is not the case would already have been culled in the frustum culling code, however, we still need to check this because some triangles may intersect the W = 0 plane and could get incorrectly culled as that breaks the test, which is quickest done by checking the smallest W against zero. The math used to check the triangle faceness can be derived by dividing X and Y with W for all points to get 2D coordinates, take the direction from the first to the second vertex, rotating that 90 degrees by swapping X and Y and reversing the sign of Y to get the edge normal, and then check the third vertex against this normal to find the winding. Simplifying that and taking out the divisions by multiplying on both sides of the equation we come up with the math above.

Pixel shader

Don’t return float4 if not necessary

In PS3.0 shaders and earlier you always had to return a full float4 vector. The shader would not compile if you tried to return anything less. This is of course wasteful if you are rendering to a one-, two- or three-channel render target. It is also common that you don’t care about alpha, but you were still forced to write something to it. This restriction has been lifted in PS4.0, so you can now return only as many components as you need. Returning a float3 instead of a float4 can in many cases reduce the number of instructions needed, especially on scalar architectures. It easily happens that one think in terms of vectors when writing shaders and do all computations in float4 even though you only care about RGB in the end. On earlier hardware this would come at only a small cost at worst. On the scalar architecture of ATI Radeon HD 2000 it could potentially waste one scalar unit for computing a value that is not used which would result in longer shaders and reduce overall performance.

Texturing

Texturing performance

Texturing is with a few exceptions required for pretty much all kinds of rendering. It is therefore of utmost importance that hardware texturing capabilities are well understood and texturing is handled properly to achieve the best performance. Modern hardware is equipped with many more ALU units than texture units as modern techniques and algorithms are more math heavy than every before. It is also the case that texture units are at the mercy of the available memory bandwidth, so equipping the hardware with more units would give diminishing returns. Given that memory bandwidth doesn’t grow at the same rate as computational power it is expected that the ALU to TEX ratio will continue to grow as it has done over the years, from the days when multi-texturing was the buzzword and hardware was equipped with two or more texture units per pipeline to today when the opposite is true. A big step

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

forward was the ATI Radeon X1900 that increased the ALU to TEX ratio to 3:1. With ATI Radeon X2900 this has increased even further to 4:1 as there are 64 shader cores and 16 texture units. With the scalar architecture with 5 values instead of 4 per pipe the actual ALU throughput has increased more from the previous generation than what that 4 versus 3 might indicate. For this reason, it is generally favorable to write ALU centric algorithms over texture heavy ones. On the other hand, the ATI Radeon 2000 texture units are much more powerful than previous generations for wide texture formats, which is important for things like HDR rendering.

One thing to keep in mind is that the quoted 4:1 ratio is the best case scenario where texels are returned in a single cycle. There are many reasons why a texture unit might operate at a much slower pace, which increases the ALU to TEX ratio even further. Things that might increase the texture sampling time include:

- Trilinear
- Anisotropic
- Wide texture formats
- Volume
- Cubemap
- Texture projection
- Lookup with gradients
- Different LOD or gradients across a pixel quad

Filtering

For a plain bilinear lookup in a 2D texture on a 32bit format the cost you pay is a single cycle. As you add trilinear that doubles to two. Mipmap clamping (such as when magnifying or sampling only from the lowest level in the mipmap chain) may reduce this to bilinear cost though, so the average cost might be lower in real-world cases. If you add anisotropic that multiplies with the degree of anisotropy. That means a 16x anisotropic lookup can be 16 times slower than a regular isotropic lookup. Don’t be too alarmed though because that hit is only taken on pixels that actually require that. For instance a large ground plane in a game would only need to take the 16x cost on the most distant pixels, and they would only be a few percent of the total pixels. Most close-up pixels, which would be the vast majority, would probably not need anisotropic at all, thus take no performance hit over trilinear. In real-world cases you can usually count on the anisotropy to be less than 2 on average.

Texture formats

The texture format has a quite large impact in texture sampling performance. On earlier generations the cost was 1 cycle per 32 bits. So all 32 bit formats and smaller (including DXT, ATI1N and ATI2N) were single cycle. 64 bit formats took two cycles and 128 bit was four cycles. The ATI Radeon 2000 is more powerful and can now do a bilinear lookup in a single cycle on RGBA16F texture. RGBA32F is two

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

AMD Smarter Choice

cycles. Note that the fixed point RGBA16 still is two cycles though, unlike the floating point format of the same size.

Cubemaps and texture projection

Another difference in the ATI Radeon 2000 architecture versus the previous generation is that the texture units have offloaded some work to the ALU for certain lookups, which makes sense given the amount of ALU power that is available. On ATI Radeon X1000 series cubemaps and projected lookups did not incur any extra cost. On ATI Radeon 2000 they come at a cost in ALU, but the TEX cost remains the same. So a projected texture lookup costs two ALU instructions and a cubemap lookup costs three. These don't use all scalars though, so actual ALU cost may be lower in real world cases as other work would fit into the empty scalar slots. Also, if you use the same texture coordinates for looking up into several textures you only get the ALU cost once. Lookups in a volume texture comes at twice the cost of 2D textures. This is true on both current and previous generation hardware.

Gradients

When doing texture lookups with gradients an additional two cycles are added. This means a regular bilinear lookup that normally takes one cycle takes three with gradients. On the previous generation you took this two cycle hit on every lookup. On ATI Radeon 2000 gradients can be stored across lookups, so if you do a lookup with the same gradients again on the same sampler, you don't take the two cycle hit. So a gradient lookup in a loop will not come at a significant cost compared to a regular lookup as long as you pass the same gradients. This is important for instance for Parallax Occlusion Mapping.

Depth & Stencil efficiency

One of the most important things to keep in mind during development is that rendering should be done with Hierarchical-Z and Early-Z techniques in mind. This was true on previous generations and continues to be true on this generation. A good use of depth and stencil optimizations can allow the hardware to cull large chunks of redundant rendering before the pixel shader, which can improve performance substantially.

The Z-culling abilities in the ATI Radeon HD 2000 series are more powerful than ever. Some of the improvements include the addition of Hierarchical-Stencil and enhancement to Early-Z so that it remains operational in almost all cases. Another important improvement is the ability to store the Z subsystem masks and information in video memory whereas previous generations had a fixed size on-chip buffer. This means that unlike previous generations all Z and stencil optimizations will remain operational for all depth-stencil buffers regardless of how many depth-stencil buffers you create. For more in-depth information about the Z and stencil optimizations see the *Depth in-depth* paper on the subject. Here we repeat the most important things to keep in mind.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

Render in front-to-back order and/or use a Pre-Z pass

This is one of the most important things to remember to get a good speedup from Z-culling. For anything to be culled it has to be occluded by previously rendered surfaces. Thus if you render in back-to-front order you will see no performance improvement, whereas front-to-back rendering can approach the performance of no overdraw. If the overdraw factor is for instance 4 on average in a particular scene the front-to-back ordering could potentially be up to 4 times faster than back-to-front. It's not necessary to do extremely fine-grained sort, often a per-object sort can go a long way. For a decent sort within objects a tool like Tootle can be used. Depending on the complexity of the scene it may be worth considering using a pre-Z pass where only the depth is rendering to lay out the scene depth in the depth buffer. This way pixel shading will essentially be done with no overdraw. However, the pre-Z pass itself is not for free, so for very simple scenes it may not be worth it. Also, it's not necessary to render the entire scene in the pre-Z pass, only objects that are major occluders need to be included. Even though the pre-Z pass does not come with any pixel shader processing it is still worth it to do some rough object based sorting for the pre-Z pass itself. If a pre-Z pass has been executed though, it is not necessary to sort the scene for the main rendering passes. In that case it's recommended that the main rendering pass be sorted according to shaders and textures to reduce state change overhead.

Avoid shader depth output

Writing depth from the shader is generally discouraged. Shader depth output causes Hierarchical-Z and Early-Z to be disabled because the shader has complete before the depth test can be done, so no depth-stencil optimizations can be used to accelerate such passes. Additionally, shader depth output can reduce performance for subsequent passes as well since shader depth output interferes with depth compression. You should consider alternative approaches to solve the problem. If it is just for killing pixels, it is more efficient to use discard in the pixel shader. If shader depth output is really necessary, consider moving the passes using it as close to the end of the frame as possible to avoid performance impact on other passes.

Draw the skybox last

Many games are still rendering the skybox first. This used to be fastest several generations ago because you could fill the color buffer directly without doing the depth test. However, on modern hardware you should render the skybox at late in the frame as possible. Only transparent objects blended against the skybox may be rendered after it. The reason for this is that the skybox itself tends to be occluded by large parts of the scene. If you render the skybox first you're shading a lot of pixels that will be overwritten later on. This is a waste of pixel processing and bandwidth. By rendering the skybox in the end only the necessary pixels will be shaded. When rendering the skybox last you may want to peg it to the far clipping plane to avoid any issues with the skybox cutting into the scene. This can easily be accomplished either by copying the fourth row of the MVP matrix into the third so that Z and W evaluates to the same value and hence Z / W becomes 1.0. Alternatively, the W value can be copied into Z in the vertex shader.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

Smarter Choice

In a similar fashion objects that are known to be in front of the scene should be drawn first. This includes the gun of the main character in a typical FPS game and opaque parts of the GUI. These may cover a surprisingly large percentage of the screen pixels.

Dealing with the small batch problem

The "small batch problem" is something you hear about increasingly more frequently these days. A number of factors has driven us into a situation where the software overhead for rendering has become a significant problem so that if the scene is rendered in a naïve way the API calls easily becomes the bottleneck and performance is significantly lower than what one might expect. One reason for this is the ever increasing GPU performance. CPU performance has of course also increased, but it has happened at a much slower pace. It was easier to be bound by the GPU in the past, so software overhead was essentially hidden. Another reason is that GPUs support an ever increasing number of features, which makes drivers more complex and adds overhead. In addition, games and applications strive to produce ever more detailed scenes which often results in more API calls.

A typical batch-limited D3D9 game will be bottlenecked by the number of draw calls or setting shader constants or both. To achieve good performance a standard rule of thumb is that you should not issue more than a few hundred draw calls. However, due to limitations in D3D9, you need separate draw calls as soon as you change any form of state, including textures, shader constants and other attributes that often varies with each object or material, so chances are you will quickly exhaust the draw call budget. Furthermore, if you need to render the scene to a render target or even a cubemap the problem multiplies. Some of these shortcomings have been addressed by Microsoft in D3D10 and by an improved driver model in Vista.

Taking advantage of D3D10

Just porting an application from D3D9 to D3D10 does not necessarily make all small batch problems go away. If you're doing all your rendering in D3D9 style, chances are you might still be API bound. Here are a few areas to look into.

Constant buffers

In D3D9 you have a fixed register set for shader constants. Before rendering with a shader you need to set the constants it is using. The problem with this approach is that the register set is small and once you set a constant to something else your previously uploaded data is permanently lost. As a result, the same data may be repeatedly uploaded to the hardware because it could not easily be preserved across passes or frames. In most applications a lot of shader constants are either static per session or static per frame, so with a better programming model were data could be preserved a lot of shader constant calls could be eliminated and the amount of data transferred across the bus each frame be reduced. D3D10 changes the

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

way shader constant are handled by introducing constant buffers. These are buffers that are conceptually pretty much identical to vertex buffers, except they hold constants. In fact, you create constant buffers and vertex buffers using the same CreateBuffer() call in D3D10, the difference is just what flags you pass in. You can create any number of constant buffers and instead of reuploading data you can often just switch constant buffer.

A plain port from D3D9 to D3D10 will probably not improve the shader constant situation much, like if you're doing things in D3D9 style and just call Map()/Unmap() or UpdateSubresource() instead of D3D9's SetPixelShaderConstantF() and SetVertexShaderConstantF(). What you need to do is analyze your application and restructure it in such a way that you can keep data on-chip without uploading it again. Don't just create one constant buffer per shader, or worse, one global constant buffer mirroring the D3D9 register set that you're updating all the time. Especially if you only partially update a buffer since that is very slow. Always upload the entire buffer when you update data and always supply the D3D10_MAP_WRITE_DISCARD flag to the Map() call. If you frequently find that you only need to update some part of it, consider separating you data into different constant buffers. But keep in mind that using too many constant buffers in a shader could be detrimental to performance, so try to keep the number of constant buffers reasonable. Also group your data logically so that it makes sense to update an entire buffer at once. Also keep in mind that constant buffers in D3D10 are essentially memory buffers rather than registers like in D3D9, so for best performance it's recommended that you also group data according to access pattern for best cache coherency. Try not to access data from several different constant buffers at once in a shader.

Texture arrays

One of the new features in D3D10 is texture arrays. A texture array is a single resource that consists of a series of textures, each with their own mipmap chain. It is different from a 3D texture in that there is no filtering across slices, and mipmaps don't shrink in Z-direction, so the number of slices doesn't change.

Texture arrays are useful to reduce draw calls and state changes. With a texture array you can easily switch material in-flight instead of having to resort to issuing a new draw call because you need to change the texture. With texture arrays you can simply access the texture you need from the array. This is great in combination with instancing. Each instance can now have its own texture and still be rendered in a single draw call, you can just use SV_InstanceID to select slice.

Even though you don't need a separate draw call for each material when using texture arrays it is still a good idea to arrange your rendering such that accesses to the texture array is roughly sorted. Just like spatially close accesses in the UV plane is best for texture caching it is equally important to try to keep as many subsequent accesses together on the same slice. For instance one might implement an instancing-like algorithm in the geometry shader that outputs each triangle of a model to several different locations. If each instance has a different texture slice this could result in very scattered accesses with poor texture cache utilization.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
Smarter Choice

Sampler states
In D3D10 the sampler states have been decoupled from the texture units. This means for instance that you can sample the same texture with different filters in the same shader. You don't need to bind the texture to two different texture units. The big advantage though is that you can now create a number of commonly used sampler states and reuse with many texture units. In many cases a few of the most commonly used sampler states could be bound on for instance the first 8 sampler state slots and stick around across most of the frame or even across the entire run of the application. This could reduce the number of sampler state changes significantly.

General optimization

Clear
Clearing color and depth buffers is important to keep hardware optimizations alive. In the past it was not uncommon to dodge clears through various clever tricks. This tends to be counterproductive on modern hardware though. Depth-stencil buffers should always be cleared with a call to Clear() in D3D9 or ClearDepthStencilView() in D3D10. This resets all internal data associated with HyperZ, which allows it to operate at full efficiency. If the buffer were to be cleared by rendering primitives into it the efficiency would be reduced over the frames. The same applies to color buffers, in particular multisampled ones. The hardware applies various compression techniques to reduce the bandwidth required to read and write color and depth-stencil buffers. Clearing ensures that the related hardware states don't go stale and reduce efficiency.

Important to note is that even though clears are important you should only clear buffers if needed. If you are not rendering into a buffer on a certain frame you don't need to clear it. For some types of simple rendering, such as updating a render target with a fullscreen quad, a color clear may not be necessary.

Instancing
Instancing is a very useful tool for reducing the number of draw calls. This is especially important in D3D9, but while the per draw call overhead is lower in D3D10 it is still important to keep number of draw calls down. The ATI Radeon™ HD 2000 series was designed with native support for instancing. This means that instancing should always be faster compared to rendering the same objects with multiple draw calls.

An important tool D3D10 gives you is the built-in SV_InstanceID value. It gives the shader knowledge about which instance it is currently rendering. On SM 2.0 hardware an instancing-like technique was sometimes implemented using vertex shader constants to store instance data. While this often didn't cut down the draw calls to one it significantly reduced the number of draw calls necessary. On the other

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

AMD Smarter Choice

hand it required multiple copies of the model to be stored in the vertex buffer. With the additional flexibility of D3D10 you can now implement the equivalent technique without multiple copies and with larger number of instances per draw call. Also, you no longer need to store an instance ID in the vertex buffer since that is generated automatically for you, which saves memory and bandwidth. Using SV_InstanceID you can index into an array of instance data in the constant buffer. Like with vertex texture fetch as discussed earlier in this paper this can result in better performance because it streams data from two different sources, one using the vertex fetch hardware and one using the constant fetch hardware. Since there are limited number of units for both you may improve your input rate if you stream from both sources as long as there is enough bandwidth to feed them both. If you are accessing per-instance data in any more fancy fashion than straight indexing from the instance ID it may be faster to use texture accesses instead since constant buffer accesses are coherency sensitive.

Über-shaders

An über-shader can be loosely defined as a shader that does more than one thing. In a traditional shader library you often have one shader for each specific task. One problem with that is the exponentially growing number of permutations. The other problem is that you need to change the shader and issue a new draw call for each particular permutation that you wish to render with. Über-shaders can solve that problem by being capable of rendering many different permutations without changing the shader or issuing a new draw call. It can be great in combination with instancing and texture arrays.

There are pitfalls with über-shaders though. If draw calls, state changes or the number of shader permutations are not an issue in your applications chances are that you would be better of sticking to a traditional approach. Über-shaders are longer and more complex. The number of instructions that needs to be executed typically increases a bit even if you use branching, as does the number of temporaries consumed by the shader. The number of temporaries is important because the fewer temporaries a shader uses the more threads the hardware can keep alive simultaneously. The more threads that are active the better the hardware can hide latencies, which is very important for instance for texture fetch efficiency. Also, if you do use über-shaders moderation is the key. If a shader becomes extremely long, for instance several thousand instructions, instruction throughput will go down as it won't fit in the shader cache.

Texture atlases

While D3D10 introduces texture arrays that solve many of the problems that were previous typically solved with texture atlases it doesn't mean atlases are no longer a good solution, especially if you are still on D3D9. If you already have a working texture atlas solution, there is no particular reason for why you should port that to texture arrays when transitioning to D3D10.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.
AMD Smarter Choice

In some cases you may want to use a texture atlas even in D3D10. D3D10 texture arrays are restricted to the same dimension for all slices and there are many cases where you might want to use textures of different sizes. In such cases a texture atlas may be a better solution.

Top 10 optimization hints

Parallelize your code

In order to utilize the power of the scalar architecture it is important that the code is parallel. Avoid unnecessarily serializing your instructions and use parentheses to introduce explicit parallelism where possible. See the *Parallelize your code* section in this document.

Optimize all shader stages

On a unified architecture it is not just the dominant shader that dictates your final performance, but all shader stages consume resources from the shared pool of computation power. The gain will naturally be larger by optimizing the most heavily loaded shader, but improvements to the less loaded shaders will still improve performance. So unlike in earlier hardware generations a pixel shader limited case might see performance improvements by optimizing the vertex shader. See the *unified architecture* section.

Make proper use of Z optimizations

Render your scene in rough front-to-back order or use a Pre-Z pass. Draw your skybox last. Draw your main character gun, opaque GUI or other front-most objects first. Avoid shader depth output. See the *Depth & Stencil efficiency* section.

Use vertex texture fetch

Getting data into your vertex and geometry shader is not only a question of memory bandwidth, but the fetching instructions may also be a limiting factor. By using vertex texture fetch you could potentially double your input rate by utilizing two separate fetching mechanisms. Splitting the data roughly equally between the vertex buffer and a texture often improves performance noticeably. See the *Vertex texture fetch* section.

Use culling in the geometry shader

The geometry shader is typically limited by the output. If it can be quickly determined that a triangle is outside the frustum or that it is back-facing you can usually achieve a significant performance improvement by not writing it out. For instance in a render-to-cubemap case most triangles need only be written to one face thus may cut down output by almost a factor of six. See the *Use frustum and back-face culling* section for details and example code.

Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

AMD Smarter Choice

Minimize geometry shader I/O

The geometry shader is typically limited by output. Input may also matter in many cases. By keeping the input and output data small you can see significant performance improvement. Packing data or trading GS output for a few instructions in the pixel shader is typically beneficial. See the *Keep data small* section.

Use instancing

While D3D10 has improved things it continues to be the case that the number of draw calls can be a significant limitation to performance. It is therefore a good idea to design your application around instancing. D3D10 makes instancing better than ever with an improved interface and tools like the SV_InstanceID system value. See the *Instancing* section.

Use the right data types

Don't use vectors when a scalar is enough. Don't compute alpha if you only care about RGB. Avoid excessive type conversions. See the *Use the right data type*, *Avoid mixing types* , *Scalar ALUs*, and *Don't return float4 if not necessary* sections.

Use dynamic branching

Dynamic branching can be used to avoid doing unnecessary work, such as computing lighting for parts of a scene that is in shadow. Good use of dynamic branching can provide a significant performance increase. See the *Dynamic branching* section.

Use constant buffers in D3D10 style

When porting a game or application from D3D9 it is important to not just directly translate D3D9 calls into equivalent D3D10 calls. If you are uploading as many constants as in D3D9 you didn't really gain anything. Try to keep as many constants around in video memory and only keep updating truly dynamic constants. See the *Constant buffers* section.

http://ati.amd.com/developer
Copyright © 2007 Advanced Micro Devices Inc. ATI, the ATI logo, CrossFire, Radeon and The Ultimate Visual Experience are trademarks of Advanced Micro Devices, Inc.

Smarter Choice