WHITE PAPER AMD

Optimizing DirectX 9 Shaders for Software Vertex Processing

July 2003

Kent Knox
ADVANCED MICRO DEVICES, INC.
One AMD Place
Sunnyvale, CA 94088

DirectX Background

In December 2002, Microsoft released to the public the latest version of their popular graphics API, Microsoft DirectX 9.0. DirectX is a software library and device interface commonly used by software developers to implement games for Microsoft® platforms. It can also be used for non-gaming related purposes, including screensavers and scientific visualizations.

Since DirectX 5, AMD worked with Microsoft to modify the DirectX runtime for optimized performance on AMD platforms. These optimizations have mainly focused on two different vertex rendering pipelines: the fixed function pipeline and the programmable shader pipeline. From DirectX 5 through DirectX 9, AMD provided an optimized fixed function pipeline utilizing AMD 3DNow!™ technology for fast floating point performance. Imagine the fixed function pipeline as being a very long machine or engine split into stages, each of which takes pre-transformed data as input, and produces transformed data as output. Each stage of the machine has a defined task, such as position transforms, vertex blending, spot lights, cone lights, texture transforms etc. Also, each stage of the machine has parameters that can be set at the API level, through `renderstate` calls. All too often, the user sets the parameters of different stages of the machine to do nothing to the data. If there are 50 stages on a machine, and only 20 are commonly used, the user may experience inefficiencies.

DirectX Transformed

When Microsoft released DirectX 8.0 to the world, they introduced a revolutionary new way of programming real-time graphics. Instead of having a monolithic engine that transforms game data for the user, why not build a custom engine out of only the stages for which the user needs work to be done? Microsoft, influenced by pioneering work done in non-real time graphics (as found in RenderMan) and with significant help from video card vendors such as NVIDIA and ATI, introduced into DirectX 8, a way to customize a vertex engine to transform and light the programmer's vertices as needed. In effect, a new language was created, a computer language for video cards called *shaders*. There are two different classes of shaders included in DirectX 8, vertex and pixel shaders. Each class of shader represents the types of operations that a programmer wants to accomplish, per-vertex and per-pixel effects. Per-vertex effects include object translations, rotations, per-vertex lighting and vertex blending. Per-pixel effects include bump mapping, displacement mapping and complex texture blending operations. Only vertex shaders run reasonably well in software; per-pixel effects are currently too data intensive to run efficiently without specialized, massively parallel hardware such as what a video card provides. When the lone word "shader" is encountered in the paper below, it refers to vertex shaders unless otherwise noted.

Shaders are a method of programming the video card. At the conceptual level, they are a programming language (assembly, and more recently higher level languages) for the graphics processor. However, only recently released video cards are designed to understand the language of shaders, while older cards do not.

Question:

What happens if a consumer buys a game using shaders (such as a GeForce 2 or Radeon), but the video card doesn't understand them?

Answer:

Microsoft's DirectX provides a software fallback where the shaders will be executed on the processor rather than the video card. AMD developed an optimizing Virtual Machine to Just-In-Time (JIT) translate the shader to optimized code for AMD processors.

DirectX 8 introduced vertex shader versions 1.0 and 1.1. These shader versions were very simple and basic in that they defined mathematical operations that could operate on vertices without any flow control operations at all. That is, a v1.1 shader could not contain any procedures, loops, or conditional constructs. The flow-of-control of these shaders is sequential in nature. Shaders are very powerful in that they allow the programmer to define/program exactly what computation s/he wants done to their vertices and to reduce any extraneous code from the fixed function pipeline. However, the fact that shaders did not have any form of flow control can be limiting in some respects. Programmers often have to code up several shaders to account for varying numbers of lights and different numbers of weights in their blending code.

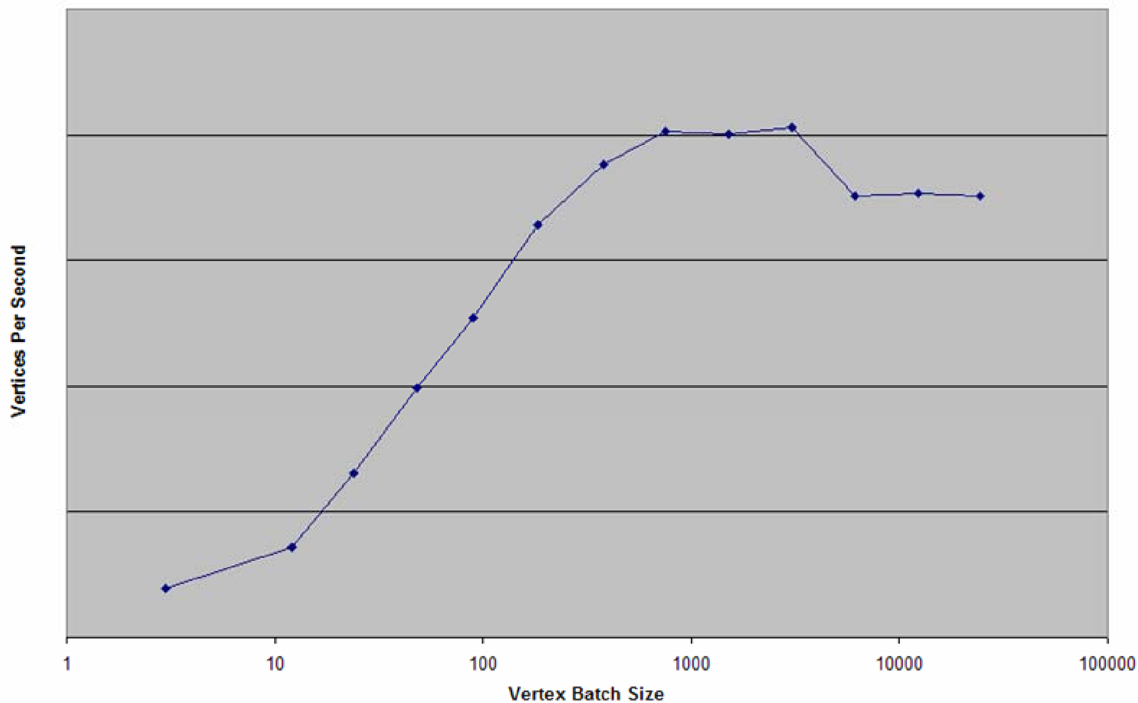
With DirectX 9, Microsoft introduced flow control instructions to make the shader language more versatile and powerful. DirectX 9 adds vertex shader versions 2.0, 2.x, and 3.0 to the shader programmer's repertoire. Version 2.0 vertex shaders add static flow control instructions and procedures. Static flow control instructions are branches that are always the same throughout the entire batch of vertices that are being processed. Usually, the branch references some kind of constant that gets set at the API level. The fact that these branches are always predicted to go the same way is important to optimization, as will be explained below. Version 2.x and version 3.0 vertex shaders introduce dynamic flow control instructions, or per-vertex flow control, where branches can branch in different directions for each vertex. Version 3.0 has all the features of 2.x in addition to expanded register sets and a few new API routines.

Just as there are efficient and inefficient ways of writing a program, there are efficient and inefficient ways of writing shaders. For the most part, techniques for writing shaders efficiently for hardware apply to writing efficient shaders for software. When there are differences, they will be noted below. Why would a person want to write efficient shaders for a game? Games are real-time software. Optimizations enable more complex environments to be displayed and simulated, as well as permit lower end platforms to play the game in the first place. The benefits are apparent and outlined below.

This paper includes a list of optimizations that will help speed up the software shader execution times. Most of these suggestions should work on hardware too, unless noted otherwise. These optimizations do NOT apply to pixel shaders, as pixel shaders do not have the option of running on the processor.

1. **Batch up vertices! Send as many vertices down the pipeline as possible in one `DrawPrimitive*` call.**

This is probably one of the oldest optimizations in the optimizer's cookbook, but it is still one of the most beneficial. This optimization applies both to hardware and software. There is a certain penalty paid on a call to the API. Think of it as a fixed cost that is not cheap. DirectX has to maintain a large amount of program state on each call to the API. In order to mitigate this fixed cost as much as possible, batch up the vertices wherever possible, and send that big batch down the pipeline. That way, if one has to pay the cost of the call to the API, at least more real work is going to be done per call. According to analysis, any batch size that is over 800 vertices is going to be using AMD's pipeline close to maximum efficiency.



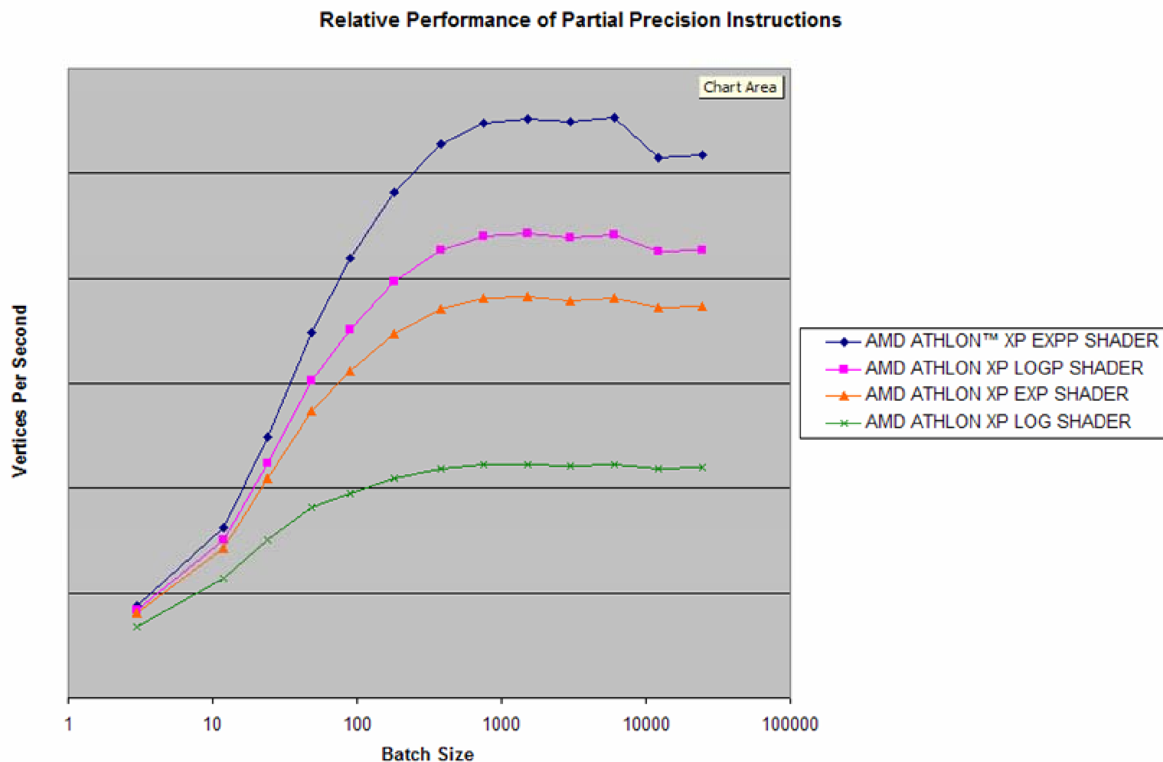
2. **Starting with DirectX 9, AMD software vertex processing is SMP aware and is designed to spawn off the appropriate number of helper threads depending on the hardware detected.**

Starting with DirectX 9, multiple processors are supported on AMD platforms! This is a feature that only gets enabled when the shader vertex virtual machine (VVM) detects more than 1 processor present in the system. It is worth mentioning that SMP and splitting batches involves a certain amount of overhead and the VVM does not bother utilizing the second processor unless a certain threshold of work is reached in the batch passed down to it. The threshold is measured as a function of the size of the individual vertices and the number of vertices in the batch. When that threshold is exceeded, the batch is split and each sub-batch is sent to a

separate processor. AMD has found that given the typical vertex size, the average number of minimum vertices needed to take advantage of the second processor is somewhere around 700. This is another example of why it is good to batch up vertices into a single batch, because maybe five little batches will not take advantage of the second processor, but one large batch will.

3. Use the partial precision instructions for lighting calculations instead of the full precision instructions.

DirectX provides partial precision `exp` and `log` instructions. As defined by the specification, the `expp` and `logp` instructions only require 10-bits of precision instead of 21-bits. This saves a significant amount of cycles in the software implementations (refer to chart below). If the result of the instruction is being used to calculate lighting, 10-bits of precision is typically enough. The naked eye will have a hard time trying to distinguish the difference, especially for fast, dynamically changing scenes. When compiling shaders with high-level languages, such as HLSL or Cg, remember that the compilers usually default to full precision instructions. Each compiler has a way to force itself to generate partial precision instructions either through command line parameters or individual flags (refer to the documentation included with the compiler on usage).



4. Use the macro instructions whenever possible. Use them appropriately.

This technique applies especially to software performance but not so much to hardware. For software, when the VVM sees an `m4x4` instruction, it knows that the program wants to do a matrix multiplication and it has special codepaths to make that as efficient as possible. On hardware, most drivers will decompose the macro instructions into their composing instructions but this, of course, depends on the hardware vendor and the driver. The trick is to make sure that the appropriate macro instruction is called for the work that needs to be done.

For example, if a position point needs to be translated/rotated, most of the time an `m4x3` instruction will suffice instead of an m4x4 instruction. The `m4x4` instruction executes one extra `dp4`, which is useless work if one doesn't need it. In the worst case, using the macro instructions should be no worse than writing out the macro into its composing instructions.

Instead of writing:

```
dp4   r0.x, v1, c[0]
dp4   r0.y, v1, c[1]
dp4   r0.z, v1, c[2]
```

use the macro:

```
m4x3  r0, v1, c[0]
```

5. Writemasks matter! When there are components that don't need to be used or initialized, don't write to them.

Destination registers can take a mask called the writemask that informs the virtual machine which components to care about in an instruction. These writemasks are important because they instruct when it is okay to skip extra computation. For example, consider this simple instruction:

```
add   r0, v0, v1
```

This is in fact 4 addition operations, 1 each for the 4 separate components `x`, `y`, `z` and `w`. However, if all that the shader cares about is the `x` component, then the instruction should be written like this:

```
add   r0.x, v0, v1
```

That lets the VVM only perform a single addition, for the `x` component. In practice, the VVM does a thorough job of analyzing the shaders it receives and only processes those components that are important to the shader.

6. Be aware that processor hardware does not support 16-bit floating point numbers natively!

DirectX 9 has introduced a few new data type formats to help developers specify to the DirectX runtime what type of data they are passing in. These new formats include the `float16` data

types. This new data type was introduced to help graphics card vendors pass data along the AGP bus to the card and to reduce register pressure inside the card. A 16-bit float is half the size of a 32-bit float, so twice the amount of data is available within the hardware registers and it can dramatically reduce the size of the batch that is being passed to the card, thereby speeding up AGP transfers. Texture coordinates are usually specified with a 32 bit floating point value which is overkill for most applications as 16 bits usually provides all the resolution that one needs. For a video card, minimizing used register space and AGP performance is very important, but the processor has different concerns.

Modern processors do not have native support for 16 bit floating point numbers, they can only handle 32 bit floats (or 64 bit doubles). When the shader VVM gets passed a batch containing 16 bit floating point numbers, it has to expand the data to 32 bits before it can process them. This is not a simple process, and is extra computation that would not otherwise need to be done for software implementations. This automatic conversion of 16 bit to 32 bit floats nullifies any register savings the developer gets inside of the VVM, and software implementations do not care about AGP performance because it gets the data from system memory. The software VVM would much rather have the bigger batch sizes than the shortened bits, because it will have to expand the floats back into the original 32 bits to process them anyway.

7. The `texldl` instruction introduced in vertex shader 3.0 may be slow in software vertex processing.

Vertex shader vs.3.0 introduces the `texldl` instruction, which allows the shader to sample a texture with a specified LOD. This instruction is not optimized in the AMD vertex machine, and when the VVM encounters the instruction, it reverts back to a Microsoft reference implementation. The results of the instruction will be correct on AMD's implementation, but the developer will notice a slowdown.

8. Conditional branches (`ifc`, `ifp`, `breakc`, `breakp`) are VERY expensive! Use them sparingly.

With the introduction of vertex shader 2.x and vertex shader 3.0, shader programmers now have the ability to "on the fly" compare a conditional expression and branch according to the outcome of the expression, such as `{ if_lt v0.x, v1.x…. }`.

Something that may not be obvious to the shader writer is that these conditional expressions are expensive in software vertex processing, much more expensive than an equivalent static branch on a constant, which were introduced in vertex shader 2.0. This is a consequence of the architecture of the virtual machine, which is SIMD 4 wide. It processes 4 vertices at a time, and when it encounters a conditional branch in the code, it has to take into account the possibility of a split batch and some of the vertices going into the if block and some going into the else block at the same time. This introduces extra housekeeping code that is necessary but may slow the performance of the shader down.

If there is a possibility that the shader programmer could calculate ahead of time the result of the conditional expression, place the answer in a constant register and use a static branch instruction such as `{ if b0 … }`, that could really speed the performance of the shader up. This same optimization applies to the `breakc` and `breakp` instructions. Calculate any

conditional expressions at all possible outside of the shader and re-write the shader to use the batch-constant branches. The goal of this optimization is to remove any possible variance outside of the shader as is possible because that undermines SIMD performance.

However, it is worth noting that if the branch is truly per-vertex and not per batch, and the branch is being used to skip a LARGE section of code (such as maybe an early-out conditional for a cone light), then the per-vertex branch may still be worth it. It depends on the cost of the housekeeping code vs. the savings the branch may give. There is truly only one way to find out, and that is to experiment and benchmark the application. The message that this optimization tip is trying to get across is to use a static branch on a constant instead of a per-vertex branch whenever possible, because the VVM will be able to take better advantage of the static branch.

9. Use the `a0` register as little as possible. This is a speed bump in calculations.

The `a0` register is a potentially useful register that effectively allows random access to the constant register file. The problem for software implementations is that it has to treat that register as being random at compile time. Of course, there are times when this register must be used, such as indexed vertex blending, but if the shader can save the results of a previous `a0` reference, it is worth the register to save the value. Here is an example:

Unoptimized code:

```
dp4   r0.x, v0, c[a0.x+1]
dp4   r0.y, v0, c[a0.x+2]
dp4   r0.z, v0, c[a0.x+3]
dp4   r1.x, v1, c[a0.x+1]
dp4   r1.y, v1, c[a0.x+2]
dp4   r1.z, v1, c[a0.x+3]
```

Optimized code:

```
mov   r2, c[a0.x+1]
mov   r3, c[a0.x+2]
mov   r4, c[a0.x+3]
dp4   r0.x, v0, r2
dp4   r0.y, v0, r3
dp4   r0.z, v0, r4
dp4   r1.x, v1, r2
dp4   r1.y, v1, r3
dp4   r1.z, v1, r4
```

In effect, this is adding three instructions to the code to get the same result, except the results of the `a0` references are cached inside of the temporary registers. That way, the VVM does not have to calculate the contents of the `a0` reference again. The reason that the `a0` reference is so slow is because the value inside of the `a0` can be different for every vertex, and the VVM cannot properly predict at compile time what register it may be accessing.

Summary

The optimization tips listed above should help the shader developer stay clear of any speed tar pits s/he may encounter while developing their shaders. For the most part, the rules for optimizing vertex shaders for software transformation and lighting are the same rules for optimizing vertex shaders for hardware transformation and lighting, with a few notable exceptions. One general-purpose optimization tip left out above that should work for ALL hardware is to eliminate instructions that are not needed or redundant. In other words, the shorter the program the faster, and don't calculate values twice if not needed. This includes moving loop invariants outside of loops too which can be an important optimization if the shader spends a significant amount of time inside of loops.

Finally, the performance of software shaders should be important to the shader developer. Many mid to low range computer solutions have "value" video cards installed in them, and some of those do not support hardware transformation and lighting. For instance, the GeForce 4 MX uses software transformation and lighting. Integrated video in unified memory architecture (UMA) systems may also use software transform and lighting to keep costs down. Also, if a computer user were to upgrade their processor and keep their video card, it may be that the game will run faster with the shader executing on the new processor than on the old video card.

For more information on Microsoft's DirectX, what the latest video cards support, Vertex or Pixel shaders, please refer to the links provided below:

http://www.microsoft.com/directx (Microsoft SDK and help docs, good starting place)
http://www.nvidia.com/ (Information on Nvidia's cards)
http://developer.nvidia.com/ (Nvidia's Developers website, lot's of information for games developers with presentations and plenty of sample source code)
http://www.ati.com/ (Information on ATI's cards)
http://mirror.ati.com/developer/ (ATI's Developers website, lot's of information for games developers with presentations and plenty of sample source code)
http://www.renderman.org/ (RenderMan is a spec for a non-real time graphics renderer. Much work on programmable shaders was started here.)

© 2003 Advanced Micro Devices, Inc.
AMD, the AMD Arrow logo, AMD Athlon and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this document are for identification purposes only and may be trademarks of their respective companies.