Shaders on Snapdragon: Best Practices

Technical Reference

80-N5422-1 Rev. 1

May 12, 2011

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Revision history

Revision	Date	Description
A	Month Year	Initial release

Contents

1 Best Practices

1.1 Compile and Link During Initialization

Compiling and linking shaders is a time intensive process, especially when compared with other calls in OpenGL ES. Load and compile shaders during initialization and invoke glUseProgram to switch between them while rendering.

1.2 Avoid Branching

Branching in shaders is a potentially slow operation and should be avoided if possible. There are three types of branches, listed in order from best performance to worst:

- Branching on a constant, known at compile time.
- Branching on a uniform variable
- Branching on a variable modified inside the shader

The first type of branch, branching on a constant, may yield acceptable performance.

As an example of how dynamic branching can be expensive, consider fragment processing. Because multiple fragments are processed at once, if some fragments take a branch while others do not, the GPU must execute both paths.

1.3 Avoid Loops

Loops can potentially hurt shader performance. Consider unrolling loops, or replacing them with a vector operation.

For intance, replace this:

```
for(i=0; i < 4; i++)
{
   diffuse += ComputeDiffuseContribution(normal, light[i]);
}
```

With this:

```
diffuse = ComputeDiffuseContribution(normal, light[0]);
diffuse += ComputeDiffuseContribution(normal, light[1]);
diffuse += ComputeDiffuseContribution(normal, light[2]);
diffuse += ComputeDiffuseContribution(normal, light[3]);
```

If a loop is absolutely necessary, consider using a constant loop count to avoid a dynamic branch.

1.4 Avoid Discarding Fragments

Alpha test and killing an instruction in the fragment shader may disable hardware optimizations. If either are necessary, attempt to render geometry which depends on them after opaque draw calls.

1.5 Avoid Modifying Depth in Fragment Shaders

Similar to the case of discarding fragments, modifying depth in the fragment shader may disable hardware optimizations.

1.6 Avoid Texture Fetches in Vertex Shaders

1.7 Precalculate As Much As Possible

Move as much code out of the shader as is reasonable; if a value does not need to be calculated for every vertex then move the calculation to the cpu and pass the value to the shader as a uniform variable. Computing values not dependent on vertex or texture stream information for every vertex or fragment is wasteful and could have serious performance implications.

1.8 Favor Vertex Shader Calculations Rather Than Fragment Shader Calculations

In a typical scene, vertex count will be significantly less than fragment count. Consequently, it is possible to reduce GPU workload by moving calculations from the fragment shader to the vertex shader. This is similar in concept to the recommendation above in the way that it eliminates redundant computations.

1.9 Favor Smaller Shaders

Smaller shaders tend to use less general purpose registers, allowing more shader threads to be in flight at one time and latency to be better hidden. On the PC many developers use an Uber or do-it-all shader in order to minimize driver calls or state changes. This should be avoided on mobile hardware. Embedded driver calls are less of an issue than their counterparts on PC, so shaders should be authored with performance as the priority. This is dependent on driver overhead and may not hold true on all platforms.

1.10 Pack Shader Interpolators

Interpolated values, or varyings, require a general purpose register to supply data to the fragment shader. All interpolators have 4 components, whether they are used or not. Better performance may be possible by eliminating empty components. A simple example would be packing 2 vec2 texture coordinates into one (vec4 by definition) interpolator.

1.11 Measure, Test, and Verify Your Results

Finding bottlenecks is paramount to optimization, whether the application in question is vertex bound, fragment bound, texture fetch bound, etc. Measure performance before any attempt at making the code faster. Use tools such as the Adreno Profiler or even software timers to take these measurements.

Do not assume something runs faster solely based on intuition. When code is modified to perform better it may disable compiler/hardware optimizations which were more beneficial. Always measure timing before and after changes to verify modifications performed for the sake of optimization did not hurt performance instead.