OpenGL ES 2.0 Performance Guidelines for the Tegra Series

Version 0.2

Contents

Introduction

NVIDIA's Tegra mobile system on a chip (SOC) includes an extremely powerful and flexible 3D GPU whose power is well matched to the OpenGL ES 2.0 APIs. For optimal content rendering, there are some basic guidelines and several tips that can assist developers in reaching their goals. This document will detail these recommendations, as well as a few warnings regarding features and choices that can limit performance in 3D-centric applications.

The 3D GPU in the Tegra series SOC contains a programmable vertex shading unit and a programmable fragment shading unit, each of which are accessible via OpenGL ES 2.0's GLSL-ES shading language. Tegra also includes a high-performance multi-core ARM Cortex A9 CPU and a high-bandwidth memory controller (MC) to round out the components of 3D rendering.

Optimal performance is achieved by:

1) Maximizing the efficient use of the fragment shading unit and vertex shading unit via smart shader programming
2) Minimizing the use of the CPU by avoiding redundant and ill-optimized rendering methods.
3) Optimizing the use of memory bandwidth across the fragment unit, vertex unit and display systems.

This document will cover aspects of all of these elements. Note that all quoted numbers are relative to common clock settings on the Tegra 250. Numbers on other Tegra variants may differ slightly.

Basic Performance Notes

In real-world applications, the most common performance bottlenecks are:

1) Fragment fill rate for applications using long shaders and/or lots of overdraw
2) Memory bandwidth on devices with large screens or when using large/deep textures
3) Lack of CPU/GPU parallelism for applications that use redundant or GPU-unfriendly OpenGL ES code

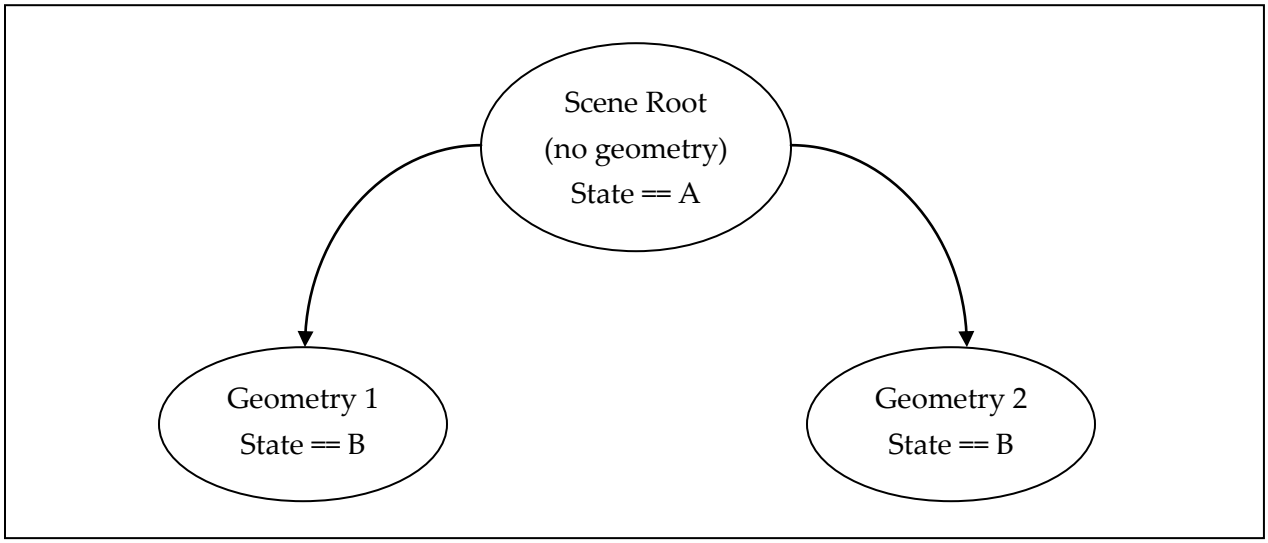
Note that the vertex shader unit is not in this list. The vertex shader unit in Tegra is extremely powerful, and is rarely the bottleneck in current mobile 3D applications (however, note the following sections on how to best keep the vertex shader unit busy).

Maximizing the GPU and CPU/GPU Parallelism

The most common initial performance issues in 3D apps tend to involve causing the driver to do needless work or doing work in the app (on the CPU) that can be better done on the GPU.

Avoid Redundant State Changes

Avoid redundant state changes to the driver (e.g. `glEnable`/`glDisable`). There are several common cases:



Do not "Push and Pop"

Do not "push" and "pop" render state e.g. during a scene graph traversal; every render state change should be directly related to a draw call. Often, push/pop-style behavior can lead to cases such as the following (see the simple scene graph above):

- Set state to an initial value A at the start of the frame based on the root.
- Set state to B and traverse down into object (driver must flag a change)
- Draw object with state B
- Step up the tree, out of the object and reset the state to A (driver must flag a change)

- Set state to B again and traverse down into another object (driver must flag a change)
- Draw object with state B

In this case, both objects were drawn with the driver having to at least process the changed state to determine that it hadn't actually changed – it was B in both draw calls. Associate state with drawable objects and set accordingly.

Avoid Setting Entire "Materials" on Each Draw Call

Do not send down every render state to GLES on every draw call and assume the driver will test for unchanged values. Use high-level app knowledge to send down only those that have changed, since this can often be done with far fewer comparisons at a higher level.

Avoid Changing Expensive States

Know which states are particularly expensive, and do not change them very frequently. Particularly expensive states include:

- **`glUseProgram`**: changing shader programs can be very expensive, as the shader program is responsible (according to the GLES spec) for storing and restoring the state of all of its uniforms (shader constants). The more uniforms in the shader, the more expensive swapping will be. Also, shaders are programs, and swapping them can cause significant performance issues
- **Some texture formats**: When using runtime-compiled shaders, switching between non-floating-point and floating-point texture formats used with a given shader can cause a driver-level shader change and perhaps a recompile.
- **Alpha/Pixel blending mode**: When using runtime-compiled shaders, switching pixel blending modes used with a given shader can cause a driver-level shader change and perhaps a recompile. This is one case where it may be worthwhile to have independent versions of a shader, one for each blended (and the non-blended) mode, and use a fixed blending mode with each copy.
- **Buffer masking**: When using runtime-compiled shaders, switching buffer masking modes used with a given shader can cause a driver-level shader change and perhaps a recompile

Consider State-Sorted Rendering

Where possible, accumulate your drawable objects into sets, grouped by expensive states like shader program, and render all objects with those same states together, changing state only at

the start of each different set, not each object. This form of state gathering can also be useful for analysis.

Avoid CPU-GPU Pixel Transfers

Avoid the following functions on a per-frame basis, as they use memory bandwidth and can stall the rendering pipeline, minimizing GPU/CPU parallelism:

- `glReadPixels`
- `gl*TexImage*`
- `glBuffer*Data`

Avoid CPU-processed Vertices

Processing vertices on the CPU is sub-optimal for several reasons:

- It uses the CPU for work that is better-suited to the GPU's vertex unit
- It leaves the powerful GPU vertex unit underworked
- It requires transferring the transformed vertices to OpenGL ES each frame.

Thus, it is best to rework older CPU-based vertex transforms and deformations (such as those required with OpenGL ES 1.x's restrictive pipeline) into vertex shaders. This can allow for a range of optimizations, since vertex shaders on Tegra can utilize a wide range of data types directly (float, half-float, byte, short, etc). This can allow for smaller vertex data than would have to be kept around for CPU-based vertex processing.

Maximize Geometry per API Call

For most applications, peak performance will be attained by feeding as much rendering data as possible to OpenGL ES 2.0 with the lowest number of API calls. Geometry should be sent in large batches, to ensure maximal CPU-GPU parallelism and to allow the GPU pipeline to run at maximum efficiency.

Vertex Shader Performance

Tegra's vertex shader unit is extremely powerful and flexible. It fully supports conditionals and looping. It is capable of transforming vertices at a rate of over 42M vertices per second. The most important methods of increasing vertex shader performance and utilization are to use it for as much processing as reasonably possible and to feed it well.

Optimally Feeding the Vertex Shader

- Use indexed geometry (`glDrawElements`) to maximize the re-use of coincident vertices. This optimizes the use of the vertex unit and minimizes memory bandwidth use.
- Use VBOs ***and*** IBOs (index buffers) for all geometry to maximize parallelism
- If you need to use "dynamic" (CPU-processed) vertices, put them in VBOs as well
- Mark as many VBOs as possible with `GL_STATIC_DRAW` and do not rewrite them
- Prefer fewer independent vertex attributes. Instead, pack multiple vertex attributes into fewer, wider vertex attributes (e.g. a pair of 2D UV sets packed into one 4D attribute).
- Use smaller-sized vertex attribute types where possible. Tegra supports byte and short attributes as well as 16- and 32-bit half-float and float attributes. Packing normals into 32 bits (X8Y8Z8<unused>8) can reduce vertex normal member bandwidth by 3x (3x32 becomes 4x8) with little or no reduction in image quality. This minimizes memory bandwidth use.

VBOs

Maximum performance on Tegra is achieved only when ***all*** vertex attributes and primitive indices are read from Vertex Buffer Objects (VBOs). Even dynamic geometry should be stored in VBOs. In most cases, only a few attributes of a dynamic primitive are actually dynamic (often positions and normals). These attributes should be uploaded to an interleaved vertex buffer each time the data is changed. However, the static attributes of that same object (e.g. texture coordinates, per-vertex colors) should be packed into another (static) VBO. This allows all of the vertex attributes of the dynamic object to be read from VBOs while avoiding having to reload the static vertex attributes every frame, as would be the case if a single VBO was used for all of the object's vertex attributes (static and dynamic).

When using a VBO for dynamic vertices, it is important to avoid an immediate cycle of `glBuffer*Data/glDraw*/glBuffer*Data/…` with the same buffer. Locking a VBO right after a rendering call using that buffer limits CPU-GPU parallelism and can block the app. Use at least a pool of round-robin VBOs for dynamic objects (reusing in least-recently-drawn fashion), and if possible, use an independent VBO per dynamic object. This avoids stalling the app to wait for a pending draw call using the same VBO.

Vertex Shader Guidelines and Optimizations

Keep the Vertex Shader Busy

Analyze the values that are computed in the vertex+fragment shader effect as a whole. Move per-fragment operations that are constant, linear, or "near linear" across a triangle from the fragment unit to the vertex unit. This can minimize per-fragment work, cut down on varying data elements used to communicate between the vertex and fragment unit and utilize the powerful looping and conditional support in the vertex unit.

Character Skinning and the Vertex Unit

Moving character skinning from the CPU to the GPU is a perfect way to offload the CPU and lower memory bandwidth. OpenGL ES 2.0 makes dynamic character skinning possible on the GPU even if the skinning method does not fit the "basic bone-palette" limitations. Even more complex skinning can be done on the GPU (e.g. bone skinning and morph deformations). By moving all skinning the GPU, we can also avoid using dynamic vertex buffers, since all of the source data (except matrices) can be static. However, there are a few recommendations for character skinning on the GPU:

- Analyze the use of bone matrices per object and avoid passing down unused bone matrices as uniforms for a given object.
- Analyze bone weights per vertex offline and cull bones with inconsequential weights.
- Since bone matrices are normally rigid transforms, consider using 3x4 matrices (a set of 3 4-vector uniforms) to represent each as a 3x3 rotation+scale and a 3D translation rather than 4x4 matrices for bones, especially if the bone palette is large. Then the final transform from world or post-deformed model space to clip space can be a single 4x4 matrix. This can cut the number of 4-vector uniforms per vertex shader by 25%.
- If multiple sub-sections of a character are to be drawn with the same shader, but each with different rendering settings, consider setting the shader and its bone transform uniforms ***once***, then interleave texture and render state changes with sub-mesh draw calls without ever changing the shader or the bone uniforms. This can greatly lower the overhead of the sub-object rendering. In this case, since the entire character's palette of bone matrices can be sent down once, it is fine that each subsection of the mesh does not use all of the bones.
- Carefully analyze the performance of multi-pass rendering algorithms with complex GPU skinning, since GPU skinning is computed for each rendering pass, not once per frame.

Fragment Performance

High-Level Fragment Performance Guidelines

The 3D engine clock on Tegra 250 platforms is typically set to 300Mhz. The fragment shader performance is measured in the number of fragments rendered per second, also called the fill rate. However, it is often more useful to express performance in a number of 3d engine clock cycles per one rendered fragment. The fragment shader unit can produce shaded fragments at a maximum of one fragment per 3D clock. This translates (at typical clock) to a maximum fill rate of 300M fr/s for a single-cycle shader.

It is worth mentioning that early depth and stencil tests are performed before the fragment shader with a speed of 4 fragments per clock. This means that fragments can be culled by depth and stencil tests 4x faster than they can be rendered. Fragments can be culled as fast as 1.2G fr/s with 300Mhz clock. Using depth and stencil tests is strongly recommended in application with high redraw rate (ratio of rendered fragments to modified pixels). However, note that some fragment features like conditional discard will disable early depth and stencil tests and lower the rate of fragment rejection. See the details in the later section on depth and stencil kill, as well as the section on discard.

Note that `glClear` does not use 3D engine (it uses 2D hardware instead). 550M cleared pixels per second are possible in practice, so clearing with 3D polygons is not recommended unless the scene geometry itself covers the entire screen (e.g. an interior scene).

Data-Related Fragment Guidelines

Texturing

Texture Formats

Where possible, use texture formats with the lowest number of bits per pixel that will fulfill the needs of the source artwork and its use in the shader. Low bit-per-texel formats include:

- **RGB:** DXT1, ETC
- **RGBA:** DXT3, DXT5
- **Single-channel** (luminance or alpha): `GL_LUMINANCE`, `GL_ALPHA`
- **Two-channel** (luminance and alpha, 2D vector): `GL_LUMINANCE_ALPHA`

The availability of fragment shaders makes it easier to use the single- and dual-channel textures in creative ways. These can lower an application's memory footprint and increase performance.

The basic guidelines are:

Formats	Bits per Texel
GL_COMPRESSED_RGB_S3TC_DXT1_EXT GL_COMPRESSED_RGBA_S3TC_DXT1_EXT GL_ETC1_RGB8_OES	4
GL_COMPRESSED_RGBA_S3TC_DXT3_EXT GL_COMPRESSED_RGBA_S3TC_DXT5_EXT	8
GL_LUMINANCE, GL_UNSIGNED_BYTE GL_ALPHA, GL_UNSIGNED_BYTE	8
GL_UNSIGNED_SHORT_4_4_4_4 GL_UNSIGNED_SHORT_5_5_5_1	16
GL_UNSIGNED_SHORT_5_6_5	16
GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE	16
GL_LUMINANCE, GL_HALF_FLOAT_ARB	16
GL_RGB, GL_UNSIGNED_BYTE	32 (see note)
GL_RGBA, GL_UNSIGNED_BYTE	32
GL_LUMINANCE_ALPHA, GL_HALF_FLOAT_ARB	32
GL_RGB, GL_HALF_FLOAT_ARB	48
GL_RGBA, GL_HALF_FLOAT_ARB	64

The difference between an RGBA texture using DXT and using half-float is 8x!

Note: Tegra does not directly support 24 bit per pixel RGB textures. These are expanded by the driver at specification time to 32 bit per pixel RGBX textures. No device memory is saved with these formats, and the reformatting process at specification time requires driver work on the CPU.

Texture Filtering

Mipmapping can generally improve performance of texture-heavy applications if any of the mipmap-enabled, non-triliniear filtering modes are used for "minification" filtering. These include GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, and GL_NEAREST_MIPMAP_LINEAR.

Trilinear filtering mode (GL_LINEAR_MIPMAP_LINEAR) is more expensive than the other mipmapped modes and should be used where proven to be needed visually.

In addition, using anisotropic filtering via GL_TEXTURE_MAX_ANISOTROPY_EXT can further decrease performance and should be used sparingly; only where it is needed.

Fragment Data Types

Tegra supports two levels of fragment variable precision: fp20 (an s.6.13 floating-point format) and fx10 (two's complement s.1.8 format). Tegra can efficiently store twice as many temporaries, varyings and uniforms in fx10 format than in fp20.

- `highp` and `mediump` precision variables are both interpreted by the compiler as 20-bit floating-point values (fp20)
- `lowp` precision variables are interpreted as fixed-point 10-bit values (fx10). As fx10 can only store values of range (-2, 2), it is typically used only for color computations and normalized values (e.g. perfect for blending). Floating point precision is usually required for storing coordinates (e.g. interpolated texture coordinates).

Note: Please note that currently, the compiler falls into a non-optimal case when directly-used texture UVs are declared as `lowp`. Avoid this by keeping UVs as `highp`.

Minimizing the number of actively-used hardware vector registers at any point in a shader is important on Tegra for maximum performance. Registers are consumed by actively-used varying and temporary variables in the shader. A register or sub-section of a register can be used by several variables if those variables have non-overlapping lifespans. The shader compiler actively optimizes these. Best performance will be found by limiting the number of actively-used variables (temporaries and uniforms) at any given time. Note that a register can hold either one fp20 variable or two fx10 variables, so use of `lowp` will help maximize register usage.

Pre-Shader Fragment Guidelines and Optimizations

EGL Configurations

Certain EGL buffer configurations can cause performance degradation, including some surprising cases. The following sections detail this.

32 BPP versus 16 BPP

Note that owing to a quirk in the EGL specification, requesting a 16bpp RGB rendering buffer via `eglChooseConfig` will return 24- or 32bpp rendering configs (if available) ***before*** any 16bpp configs. Thus, it is safest to have EGL return a long (16 or 32 element) array of configs and sort/search them manually for the best-fitting config.

Note: Selecting a 32bpp config with a 16bpp screen format (or vice-versa) can result in decreased `eglSwapBuffers` performance due to the format conversion required.

Coverage Buffers

If an application includes a coverage buffer request (or for any reason uses a config that includes a coverage buffer), then buffer swapping costs can be slightly increased. Note that Coverage Sampled AA is on by default if there is a coverage buffer in the EGL config, so having a coverage buffer can lower peak fragment performance without the application setting any specific rendering states. Keep this in mind when choosing a rendering config. The EGL configuration values in question are:

`EGL_COVERAGE_BUFFERS_NV`

`EGL_COVERAGE_SAMPLES_NV`

Depth and Stencil Kill

As mentioned previously, Tegra can reject fragments via depth and/or stencil testing at 4x the peak fragment shading rate. Thus, it is best to use depth or stencil rejection when possible to increase practical fragment throughput.

Depth-Kill

Tegra can reject fragments via depth-testing at a very high rate. As a result, applications that can render opaque parts of a scene even roughly front-to-back with depth testing enabled can see a performance improvement. This is especially true if possibly-occluded objects with expensive fragment shaders can be drawn last.

If the application uses particularly complex fragment shaders with a large amount of overdraw, then even if front-to-back sorting is not feasible, the application can see higher performance using an initial depth-only rendering pass with the color buffer masking set to `GL_FALSE`. For optimal performance, applications should consider using a custom, almost null fragment shader for this pass. However, applications using runtime source-code shaders will see a performance boost by setting the color masks to `GL_FALSE` during the depth pre-pass, as the online shader compiler should substitute a trivial shader.

Stencil-Kill

Stencil-killed fragments are generally the fastest rejection cases possible, as they are 8-bit, rather than 16-bit surfaces. Stencil killing for depth complexity minimization can be more complex in terms of application setup code, and some datasets simply cannot sort geometry in this way. However, if static geometry is available pre-sorted, stencil-kill can provide maximum performance. Applications that are fill-limited and have high per-pixel fragment depth should

consider stencil-killed front-to-back rendering with depth-testing disabled. In some cases, 2D UIs done in OpenGL ES are good examples of this.

Fragment Shader Guidelines and Optimizations

Understanding Lower Bounds

There are a number of ways to approximate a lower bound on the number of clocks required to render a fragment. These can assist in optimizing shaders. We will think of the fragment shader unit in terms of a set of pipelined "sub-units" that do different fragment-related functions. The most important sub-units are the raster sub-unit, the texture sub-unit, and the ALU sub-unit. The max number of cycles between these units is a (very) rough lower bound, although obviously dependencies between the units (ALU needing a texture lookup, texture coords needing ALU computations) can raise these:

Raster Sub-Unit

The raster sub-unit can generate up to eight varyings in one cycle, grouped into four "slots". A single slot cannot hold parts of more than one varying, so it is best to group scalar lowp varyings into vector varyings!

Slot counts:

- 1 slot:
  - `lowp float` (wasteful; merge with another scalar `lowp float` into a `vec2`)
  - `lowp vec2`
  - `highp float`
- 2 slots:
  - `lowp vec3` (wasteful; merge with another scalar `lowp float` into a `vec4`)
  - `lowp vec4`
  - `highp vec2`
- 3 slots:
  - `highp vec3`
- 4 slots (a full cycle):
  - `highp vec4`
  - 2 `lowp vec4`'s

Texture Sub-Unit

The texture sub-unit can retrieve a:

- Non-floating-point RGBA texture in one cycle
- Floating-point A or LA texture in one cycle

- Floating-point RGBA texture in ***two*** cycles

The fact that the texture sub-unit generates at most one texture sample per cycle also means that there is no need to interpolate more than one directly-used texture coordinate in a single varying. This can make a difference in cycle counts. For example, the shader:

```
varying vec2 uv;
varying vec2 uv1;
varying lowp vec4 color;
uniform sampler2D tex0;
uniform sampler2D tex1;

void main()
{
        gl_FragColor = texture2D(tex0, uv) * (color + texture2D(tex1, uv1));
}
```

May take fewer cycles than

```
varying vec4 uv;
varying lowp vec4 color;
uniform sampler2D tex0;
uniform sampler2D tex1;

void main()
{
        gl_FragColor = texture2D(tex0, uv.zw) * (color + texture2D(tex1, uv.xy));
}
```

But performance testing will confirm. Consider this when packing varying values.

ALU Sub-Unit

The ALU sub-unit can execute up to four independent scalar MAD's (Multiply-Adds) per clock (actually, these are technically Multiply-Multiply-Adds with limitations), i.e.

General case:

```
x = a * b + c
```

But with some limitations on d, it can do:

```
x = a * b + c * d
```

where d is equal to b or c, modulo the free register modifiers listed in a later section (e.g. (1-d)).

Thus, there is no way to do more than 4 adds in one cycle, so the number of adds required to render a fragment divided by 4 is the lower bound on the cycle count.

However, the ALU units can do a few other related operations. Instead of 4 MADs, some other operations are possible in a single cycle, such as:

- 4-vector dot product

- 3-vector dot product (plus scalar addition)

Generally, it is only possible to do 4 multiplications per cycle. With certain constraints it is possible to do limited cases involving 6 or 8 multiplications per clock, owing to the way that the 4 independent MADs work. But in general, the number of multiplication results required to render a fragment divided by 4 is a lower bound on the cycle count.

Immediate values (numeric constants compiled into the shader) other than 0.0 and 1.0 use ALU unit cycles. While the ALU can do MADs in the same cycle as immediate values, the ALU unit can only generate three fp20 (or three pairs of fx10 or any combination thereof) per cycle and limit the number of MADs in that cycle to 3.

Multi-Unit Operations

Scientific functions (e.g. sin, sqrt) and 1/x can take more than one cycle and more than one unit and can complicate lower bound computations.

An Example

As an example, consider the shader seen earlier:

```
varying vec2 uv;
varying vec2 uv1;
varying lowp vec4 color;
uniform sampler2D tex0;
uniform sampler2D tex1;

void main()
{
        gl_FragColor = texture2D(tex0, uv) * (color + texture2D(tex1, uv1));
}
```

Analyzing it, we see:

- There are two texture loads, so the min cycle count for the texture sub-unit is 2
- There is a 4-vector color multiply, and a dependent 4-vector color add. Thus, two ALU cycles are the minimum
- There are three varying vectors needed. However, `color` and `uv1` each take two slots, thus it may be possible to compute `color` and `uv1` in the same cycle, and then `uv` in another cycle. So a minimum of 2 cycles are required in the raster sub-unit.

This indicates that across the board, this shader could require two cycles. Using a current shader compiler as of the writing of this document, the shader was compiling to two cycles.

Fragment Shader Tips and Recommendations

Utilize Free Input Operand Modifiers

Tegra can modify the values of fragment math operands "for free" in some key cases. This includes such modifiers as: (-x), (x-1), (x/2), (x*2). The compiler will apply these automatically as it can, so simply be aware of their existence. The structure is such that a given source operand can do the following (or skip them) in the given order

1. Scale by 2x
2. Subtract 1.0
3. ABS
4. Negate

Thus, (2x-1) can be a free modifier (if the compiler does not need the modifiers on the operand for another transformation…). For example, 4-component blending as follows:

```
newDest.rgb = oldDest.rgb * src.a + src.rgb * (1 - src.a)
newDest.a   = oldDest.a   * src.a + src.a   * (1 - src.a)
```

is possible in 1 cycle, as it can be written as:

```
newDest.rgb = oldDest.rgb * src.a + src.rgb * (-(src.a - 1))
newDest.a   = oldDest.a   * src.a + src.a   * (-(src.a - 1))
```

(since the MAD instructions use src.a multiple times and the (1 – src.a) can be computed from src.a via input operand modifiers)

Utilize Free Result Modifiers

Tegra can modify the values of fragment math operation results "for free" in some key cases. This includes modifiers (x/2), (2x), (4x), (clamp(0, 1)). The compiler will apply these automatically as it can, so simply be aware of their existence. One scaling followed by a clamp can be applied to the result of an operation. So the GLSL function

```
y = clamp(x, 0.0, 1.0);
```

can be free if it can be applied to the result of the previous operation. As a result, preferring clamp(x, 0, 1) to min(x, 1) if applicable (i.e. the additional clamp to zero works for the algorithm) can increase performance.

Avoid Conditional Code

Avoid conditional code in the fragment shader. Especially avoid using uniforms or other input variables to emulate discrete sets of modes. Any discrete set of modes can and should be split into a set of specialized shaders, one per each mode.

Both branches of the "if" conditional are executed to render a fragment. This can make conditional code quite long, especially when conditionals are nested.

If you need to use conditionals it is better (where possible) to express them as ternary ?: operators and GLSL functions that produce binary vectors (e.g. lessThan).

Avoid `discard`

The aforementioned 4x early Z/S optimization cannot be used if there are "discard" statements in the fragment shader. Note that this is true whether or not particular fragments will be discarded. Existence of paths that can reach discard in a shader will disable the optimization. Avoid using discard in shaders when possible. If you must use discard, consider using buffer-masking functions in GL ES 2.0 to disable buffers such as stencil or depth that need not be written.

Use uniforms instead of numerals

With a few exceptions numerical constants cannot be used directly as operands by Tegra. General immediates/numerals waste some clock cycles as well as precious register space. Uniforms, on the other hand, *can* be used directly as operands. Instead of using the following:

```
x = y * 0.23 + z * 0.75 - 0.11
```

it is often better to rewrite as follows:

```
uniform lowp vec3 c;

 (...)

x = y * c.s + z * c.t - c.p
```

where c is a uniform initialized to `vec3(0.23, 0.75, 0.11)`. This code is also more flexible and reusable than the immediate-based example if there is the possibility of needing to change these values.

The known exceptions are constants 0 and 1, which are free, as well as multiplication by specific modifiers, as discussed previously.

Use some form of Color Masking Where Appropriate

When you render to surface or use and effect that uses/requires fewer than 4 color components consider using some form of color masking. The exact method may differ from case to case as listed in this section. Either `glColorMask`, an equivalent "`#pragma`" for offline shaders, or a constant assignment of 0 in the alpha channel of the shader may be appropriate. The shader compiler should be able to save some instructions by not initializing unused output components. However, note that using the `glColorMask` or pragma methods can actually

increase memory bandwidth requirements of a non-pixel-blended shader. Thus, it is best not to mask colors with `pragmas` or `glColorMask` when `glBlend` is disabled. In these cases, it is best to simply use a constant assignment to the unused channels in the shader.

> **Note:** Avoid masking colors with pragmas or glColorMask when glBlend is disabled.

Do not update destination alpha when not needed

The blending state is often set as follows:

`GL_ADD: GL_SRC_ALPHA, GL_SRC_ONE_MINUS_ALPHA`

In this mode, the destination alpha component is updated to the target buffer but it is never used or visualized, which is wasteful of shader cycles. There are several options to minimize this expense in cases where destination alpha is unused:

> **Note:** These recommendations are ***ALL*** based upon the notion that alpha/pixel blending is being used. If the shader in question does not compute pixel blending, then the best method of optimization is to write 0 to the alpha channel in the shader (first option below).

- Writing zero to destination alpha. To do this, call `glBlendFuncSeparate` with `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ZERO`. Generally, this still requires at least one ALU sub-instruction to zero the register representing destination alpha, although in some cases it can be rolled into an existing instruction.
- Leave the destination alpha unchanged. To do this, call `glBlendFuncSeparate` with `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ZERO, GL_ONE`. This will allocate at least one extra fx10 register between the time the pixel color is loaded and until its color is written back to the surface, but will not require additional ALU instructions.
- Often the most efficient method (and the simplest) is to use `glColorMask`.

Each of these methods can end up being the most optimal, depending on the exact shader being used. In cases with a lot of blending apps should test the options.

Use Explicit access to last fragment color

The fragment shader can fetch pixel color from the render target into a variable; this is as efficient as fixed-function alpha blending. The feature is accessed via an NV extension that lets the shader access the old pixel color explicitly. The equivalent of standard alpha blending can be implemented without blending enabled in GL state with the following shader:

```
#extension GL_NV_shader_framebuffer_fetch: enable
uniform lowp col;
void main()
```

```
{
    gl_FragColor = mix(gl_LastFragColor, col, col.a);
}
```

The extension gives more flexibility for pixel blending than is possible via ES 2.0's fixed-function alpha blending. This flexibility can sometimes save a few clock cycles per fragment, since blending computation can be rolled into existing shader cycles in some cases.

API-Level Fragment Recommendations

Source-Code Shaders versus Binary Shaders

Runtime source code shaders on Tegra are extremely convenient, as they do not require the application to pre-compile their shaders, nor do they require pragmas to set the blend mode, write-masking, etc. However, there are prices to be paid for this convenience. Runtime shader compilation increases the memory footprint of the application (the shader compiler code and data structures), and significant CPU work is incurred whenever a shader must be compiled.

Shaders must be compiled or recompiled in at least the following cases:

- First rendering use of the shader program
- Each time a new, unique combination of "key" states is used with the shader program. Key states in this context include all states for which precompiled shaders require pragmas, including:
    - Color blending mode
    - Write masking
    - Texture image formats (unsigned versus signed versus half-float, not component depths or count)
    - Framebuffer format (unsigned versus signed versus half-float, not component depths or count)

In addition, using a large number of unique sets of key states with any given shader can lead to additional recompilations in some cases. The number of unique sets of key states used with a given shader should be limited whenever possible.

Memory Bandwidth

High-Level Memory Bandwidth Guidelines

On Tegra, with complex, blended, texture-heavy rendering, maximum memory bandwidth can become a performance bottleneck if care is not taken. Understanding some rough guidelines can help the developer know when they may be bumping up against such limits.

The memory interface is designed to transfer up to 8 bytes per memory controller (MC) clock cycle. This would imply about 2.5GB/s for typical settings of 333Mhz (1G = 1024^3 here). However the efficiency in real-life cases is lower, especially when multiple hardware engines access memory at once. Based on experimental data it is safe to assume between 60-90% efficiency for fragment rendering (1.5GB/s at 60% of 333Mhz). We will use this most pessimistic assumptions for these examples, in order to help form some lower bounds. Actual available bandwidth may be considerably higher than 1.5GB/s.

Memory latency (e.g. texture fetching) is well hidden on Tegra so we only need to calculate the number of memory transfers per fragment to figure the fill rate that maximizes memory bandwidth.

Examples

Here are some examples of shaders and their memory bandwidth requirements.

Example 1: Simplest Shader

We use the following shader:

```
uniform lowp col;
void main()
{
    gl_FragColor = col;
}
```

Assuming that the surface format is RGBA8888 with no depth or stencil writes, the shader writes 4 bytes per fragment so we would hit the bandwidth limit at:

(1.5 GB/s) / (4 B/fr) = 402M fr/s

Memory bandwidth will not be a bottleneck for this shader (since a 1-cycle shader would be GPU-limited to 300M fr/s).

Example 2: Simplest Blending

We use the same shader as in the first example but we also enable blending, set blend equation to `GL_ADD` and set blend function to `GL_SRC_ALPHA`, `GL_SRC_ONE_MINUS_ALPHA`.

Since the shader uses blending we need to ***read*** and write 4 bytes per fragment. Therefore, using the most conservative memory bandwidth assumption, we cannot render fragments faster than:

(1.5 GB/s) / (8 B/fr) = 201M fr/s

It turns out that the cycle count of the program itself is only one. This shader may be memory bound, since 201M fr/s is considerably lower than the GPU limit of 300M fr/s for a single-cycle shader.

Example 3: Texturing

We use the following shader:

```
uniform sampler2D tex;
varying vec2 tcoord;
void main()
{
    gl_FragColor = texture2D(tex, tcoord);
}
```

We assume for this example that we render 100x100 quads using 150x150 RGBA textures (one texture per quad) compressed with a ratio of 1:2 (2 bytes/texel, e.g. RGB565). We also use linear texture filtering without mipmapping.

We have to fetch all texture data at least once and, thanks to texture caching, only once. Therefore the average amount of texture data fetched per fragment can be calculated like this:

(150^2 tx) * (2 B/tx) / (100^2 fr) = 4.5 B/fr

We also have to write 4 bytes of color per fragment. Therefore, using the most conservative memory bandwidth assumption, the fill rate could not be higher than:

(1.5 GB/s) / (4 B/fr + 4.5 B/fr) = 169M fr/s

Since the shader is a one-cycle shader, it could run at 300M fr/s if memory bandwidth were to be ignored. But according to memory bandwidth limitations, the effective throughput is about half that (169M fr/s). Thus, the memory bandwidth limitations in this case could cause a one-cycle shader to have about the same effective throughput as a two-cycle shader (which would be GPU-limited at 150M fr/s).

The performance can be improved a lot in similar cases by changing texture size, format or filtering settings. For example, switching to DXT1 (4 bits/texel) leaves:

Tex: (150^2 tx) * (0.5 B/tx) / (100^2 fr) = 1.125 B/fr

(1.5 GB/s) / (4 B/fr + 1.125 B/fr) = 314M fr/s

Which changes the equation/balance considerably – the fragment rate crosses over to not being memory bound. In addition, keep in mind that this assumed 150x150 textures with 100x100 quads. If we increase the quad size, the amortized B/fr from texture memory reads falls, assuming caching.

Other consumers of memory bandwidth

Tegra has a unified memory; other modules compete with the 3D engine in order to access the memory.

The display engine can consume significant bandwidth when it is continuously reading the back-buffer in order to refresh the display. For example in 720P, with 60Hz native display refresh rate and RGBA8888 surface format the display engine has to transfer:

1280 * 720 * 60 * 4B = 211 MB/s

In many cases, if the application window is not full-screen then we cannot just flip the front and back buffers to swap them. We have to blit the GL backbuffer surface to a surface owned by the window manager or OS. This operation requires reading and writing 2 or 4 bytes per pixel! Swapping buffers to render a 30FPS animation in 720P window at 32 bits consumes equivalent additional bandwidth to the previous example:

1280 * 720 * 30 * (4B + 4B) = 211 MB/s

Put together, display plus blitting in a compositing OS has an overhead of ~422 MB/s.

Combined Example 1: High-quality Composited/Blended UI

If we assume the following:

- A tablet screen with 1024x600 resolution at 32bpp
- A scan rate of 60fps
- Assume a 3D-composited OS (i.e. the final rendering buffer is composited to the screen using texturing)
- A rendering and compositing rate of 30 fps in the 3D app

If we consider a tablet screen with 1024x600 resolution at 32bpp, a scan rate of 60fps and assume a window manager, then memory bandwidth for a 30fps app for scanout and compositing is:

1024 * 600 * (60 * 4B + 30 * (4B + 4B)) = 281 MB/s

We'll add into this a very expensive case of rendering: rendering each 3D pass with a full-screen-sized (1024x600) texture and alpha blending. Then each 3D fragment rendered needs to

read the frame buffer (4B), read the texture (4B, since we load one texel per fragment, caching can be ignored) and write the framebuffer (4B). That's a total of 12B of memory bandwidth per rendered fragment. So the memory cost of N frames of overdraw per frame at 30fps is:

1024 * 600 * 12B * N * 30 = 211 * N MB/s

So, with a budget of 1.5GB/s, the allowable overdraw would be about:

6 fragments per pixel = ~ 211 MB/s * 6 + 281 MB/s = ~1.4 GB/s

if we consider only memory bandwidth. But as seen above, the GPU-clock limit numbers for basic level fragment shaders are greater than this, so memory bandwidth can definitely be the limitation.

To determine specifically, the max number of cycles per fragment given 6 fragments per pixel would be

300MHz / (1024 * 600 * 30 * 6) = ~2.7 cycle shader

So the "break-even" point here appears to be about 2.7 cycles per fragment for the shader code.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com