OpenGL Performance Tuning

OpenGL Performance in a Shader-Centric World

Evan Hart – ATI Research Inc.
Overview

• Why ‘Shader Centric’?
• Performance analysis overview
• Shader Language performance tips
• Managing state
• Non-shader issues
• Multiprocessing
Why all these shaders?

- Obvious increase in visual fidelity
- Hardware design trends
- Performance
- Cool factor
Hardware Design

• Drive for new features
• Limited die space
  – Roughly 200 million transistors
• Fixed function gets bumped out
  – Collection of special shaders
  – Still highly tuned
Improved Performance

- Shaders provide direct control
  - Implement the exact algorithm
- Avoid external processing
  - Less CPU work
Why focus on shaders?

• Better
• Faster
• Cooler
Performance Analysis

- Find the bottleneck
- Balance the performance
- Rinse, lather, and repeat
Possible Pipeline Bottlenecks

CPU → Geometry Storage → Geometry Processor → Rasterizer → Shading → Frame buffer

- CPU/Bus Bound
- Vertex Bound
- Pixel Bound
Finding the Bottleneck

- Reduce the workload of different stages
- If performance does not change
  - Move on, this is not it
- If performance does change significantly
  - This is the bottleneck
  - Look to reduce the workload
Pixel Bottleneck

• Easiest to detect
  – Does performance scale with resolution?

• Multiple causes
  – Memory bandwidth
    • Disable blending
    • Reduce texture bit depth
  – Shader Performance
    • Try a simplified shader
  – Texture filtering
    • Turn off trilinear or anisotropic filtering
Vertex Bottleneck

- Harder to detect, less frequent
  - Render $\frac{1}{2}$ the triangles of each object
  - Reduce the complexity of vertex shaders
- If both scale performance
  - Vertex bottleneck
- If only reduced triangle count scales
  - Submission/fetch bottleneck
CPU Bottleneck

- Most common today (on high-end)
  - Use profilers like VTune®
  - Find API versus application time
  - If application limited
    - Make it prettier, it's essentially free
Bottlenecks

- Find major area
  - Fragment/Pixel
  - Vertex
  - CPU
- Tune to find the exact cause
Shading Language Performance

- General shader tuning
- Shader conditionals
- Marking items as const
- Reducing register pressure
- Utilizing vector ops
- Understanding the architectures
General Tuning Advice

- Remember floating point and compiler basics
- Keep hardware limitations in mind
- Balance resources
- Write clear code
Floating Point Basics

- Operations are not commutative
  - \((a*b)*c \neq a*(b*c)\)
- Limits the ability of a compiler to reorder code
- Likely to be fairly aggressive
- Do not rely on extreme reordering
Remember Hardware Limitations

• Falling to software is the ultimate performance penalty
  – Dynamic addressing not available everywhere
  – Vertex texturing is limited
Balance Resources

• Three distinct categories
  – Computation
    • Multiply accumulate
    • Transcendental operations
  – Texture
  – Interpolation

• See IHV information for more detail
Resource Balancing Hints

- Bandwidth is scarce
  - Texture fetch can absorb 32 GB/s

- Bias toward ALU operations
  - No bandwidth consumed
  - MAD cheaper than sqrt (in general)

- Using more varying almost always better
  - Moving computation back to the vertex shader
Write Clear Code

- Compilers pretty good today
- GPU compilers are getting much better
- Convoluted code may hurt the ability to optimize
Shader Conditionals

- Powerful feature
- Performance implications are complex
  - SIMD processing requires coherency
  - Can increase instruction count
  - Don’t assume it will be faster to skip work
Using Const Qualifier

• Compile time constants highly efficient
• Uniforms require JIT optimizations
• Consider specializing shaders for uniforms that take 2-3 values
  – if (doShadow > 0.0) likely inefficient
Reduce Register Pressure

- Less important than in the past
  - Compilers better
  - Hardware has evolved
- Minimize array sizes
- Avoid storing scalars in vectors
Utilize Vector Instructions

• Specifies explicit parallelism
  – Helps compiler out

• Reliably accesses ‘special’ instructions
  – DP3/DP4
  – Normalize on NVIDIA hardware
Understand the Architectures

- **Unpublished instruction sets**
  - Difficult to understand and properly tune
  - Change from generation to generation

- **ATI**
  - Vector + Scalar

- **NVIDIA**
  - Vector Superscalar

- **3DLabs**
  - Scalar
State Management

- Avoid unnecessary state change
- Avoid extra shader compiles
- Avoid frequent sampler remapping
- Minimize queries
  - Ask once and cache the value
- Test über shaders carefully
  - Switching shaders may be faster
Fixed Function Bits

- Depth Optimizations
- Use Modern Methods
  - Extensions are often performance focused
- Good state management
Depth Optimizations

• Most modern GPUs have occlusion culling
  – Can massively improve performance
  – No cost to the application
  – Simple guidelines to maximize results
Depth Guidelines

- Clear depth and stencil together
  
  `glClear(GL_DEPTH | GL_STENCIL)`

- Clear the depth buffer

- Draw roughly front to back
  - Opaque objects

- Consider a depth fill pass
  - Only if shaders are expensive

- Avoid killing pixels while updating depth
  - Alpha test or Discard

- Avoid reading the depth buffer
  - `glReadPixels` or `glCopyTexSubImage`

- Use a consistent depth function
  - Preferably directional (GL_LEQUAL)
Use Extensions

- **Most address specific bottlenecks**
  - Frame Buffer Object
    - Context switch overhead
  - Vertex Buffer Object
    - Vertex caching
  - Pixel Buffer Object
    - Simple pixel transfer speeds
State Management

- Attempt to minimize state changes
  - Toggling states costs CPU time
  - Avoid resetting to default ‘just because’
- Some changes are worse than others
  - Blend enable/disable cheap
  - Shader change expensive
  - Texture format change expensive
Preparing for Multi-CPU

• Understand OpenGL's synchronization semantics
  – Multicontext gotchas
    • When does something delete?

• Possible strategies
  – Single graphics worker thread
  – Single context multiple threads
  – Multiple contexts multiple threads
OpenGL Multithreading Basics

- Context is limited to one thread
- Limited context synchronization
  - Primarily glFinish
  - Also glBind* to a limited degree
- Easy to run into trouble
Object Lifetimes

• Delete does not mean delete
  – Other contexts could be using it
  – Synchronize when destroying an object
  – OpenGL spec is a little fuzzy here
    • Expect slightly different behavior
Single Graphics Thread

- One context working on a single thread
  - No graphics synchronization overhead
  - Services rendering requests
    - Drawing
    - Resource management
  - Other CPU free to handle physics, etc.
  - Best overall model presently
Multicontext with One Thread

- **Advantages**
  - Keep state ‘clean’ on each context
  - Synchronization logic is easy

- **Disadvantages**
  - Context switch is expensive
    - Unlikely to be successful
MultiContext/MultiThread

- Possibly allow background loading
  - Thread A renders
  - Thread B loads textures

- Synchronization is complex
  - May require glFinish
  - Driver may have to do extra internal synchronization
    - Not likely to be fast today
Multi-GPU Hints

• Minimize interframe dependence
  – Clear buffers
  – Avoid glCopy* operations
  – Avoid rendering to textures every other frame

• Simon has additional details...
Thanks

- ATI Dev Team
- NVIDIA Dev Team
- John Spitzer
  - Original pipeline slides