Practical Performance Analysis and Tuning

Ashu Rege and Clint Brewer

NVIDIA Developer Technology Group
Overview

- Basic principles in practice
- Practice identifying the problems (and win prizes)
- Learn how to fix the problems
- Summary
- Question and Answer
- Performance Lore
Basic Principles

- Pipelined architecture
  - Each part needs the data from the previous part to do its job
- Bottleneck identification and elimination
- Balancing the pipeline
Pipelined Architecture (simplified view)
The Terrible Bottleneck

Limits the speed of the pipeline
Bottleneck Identification

Need to identify it quickly and correctly
- Guessing what it is without testing can waste a lot of coding time

Two ways to identify a stage as the bottleneck
- Modify the stage itself
- Rule out the other stages
Bottleneck Identification

- Modify the stage itself
  - By decreasing its workload

  If performance improves greatly, then you know this is the bottleneck
  - Careful not to change the workload of other stages!
Bottleneck Identification

- Rule out the other stages
  - By giving all of them little or no work

- If performance doesn’t change significantly, then you know this is the bottleneck

- Careful not to change the workload of this stage!
Bottleneck Identification

Most changes to a stage affect other stages as well
Can be hard to pick what test to do
Let’s go over some tests
Bottleneck Identification: CPU

CPU workload

What could the problem be?

Could be the game
- Complex physics, AI, game logic
- Memory management
- Data structures

Could be incorrect usage of API
- Check debug runtime output for errors and warnings

Could be the display driver
- Too many batches
Bottleneck Identification: CPU

- Reduce the CPU workload
  - Temporarily turn off
    - Game logic
    - AI
    - Physics
  - Any other thing you know to be expensive on the CPU as long as it doesn’t change the rendering workload
Bottleneck Identification: CPU

- Rule out other stages
  - Kill the DrawPrimitive calls
    - Set up everything as you normally would but when the time comes to render something, just do not make the DrawPrimitive* call
    - Problem: you don’t know what the runtime or driver does when a draw primitive call is made
  - Use VTUNE or NVPerfHUD (more info later)
    - These let you see right away if the CPU time is in your app or somewhere else
Bottleneck Identification: Vertex

**Vertex Bound**
- What could the problem be?
  - Transferring the vertices and indices to the card
  - Turning the vertices and indices into triangles
  - Vertex cache misses
  - Using an expensive vertex shader
Bottleneck Identification: Vertex

- Reduce vertex overhead
  - Use simpler vertex shader
    - But still include all the data for the pixel shader
  - Send fewer Triangles??
    - Not good: can affect pixel shader, texture, and frame buffer
  - Decrease AGP Aperture??
    - Maybe not good: can affect texture also, depends on where your textures are
    - Use NVPerfHUD to see video memory
      - If it’s full then you might have textures in AGP
Bottleneck Identification: Vertex

- Rule out other stages
  - Render to a smaller backbuffer; this can rule out
    - Texture
    - Frame buffer
    - Pixel shader
  - Test for a CPU bottleneck
  - Can also render to smaller view port instead of smaller backbuffer. Still rules out
    - Texture
    - Frame buffer
    - Pixel shader
Bottleneck Identification: Raster

- Rasterization
  - Rarely the bottleneck, spend your time testing other stages first
Bottleneck Identification: Texture

Texture Bound

What could the problem be?
- Texture cache misses
- Huge Textures
- Bandwidth
- Texturing out of AGP
Bottleneck Identification: Texture

- Reduce Texture bandwidth
  - Use tiny (2x2) textures
    - Good, but if you are using alpha test with texture alpha, then this could actually make things run slower due to increased fill. It is still a good easy test though
  - Use mipmaps if you aren’t already
  - Turn off anisotropic filtering if you have it on
Bottleneck Identification: Texture

- Rule out other stages
  - Since texture is so easy to test directly, we recommend relying on that
Bottleneck Identification: Fragment

- Fragment Bound
  - What could the problem be?
    - Expensive pixel shader
    - Rendering more fragments than necessary
      - High depth complexity
      - Poor z-cull
Modify the stage itself

- Just output a solid color
  - Good: does no work per fragment
  - But also affects texture, so you must then rule out texture

- Use simpler math
  - Good: does less work per fragment
  - But make sure that the math still indexes into the textures the same way or you will change the texture stage as well
Bottleneck Identification: FB

Frame Buffer bandwidth

What could the problem be?

- Touching the buffer more times than necessary
  - Multiple passes
- Tons of alpha blending
- Using too big a buffer
  - Stencil when you don’t need it
  - A lot of time dynamic reflection cube-maps can get away with r5g6b5 color instead of x8r8g8b8
Bottleneck Identification: FB

Modify the stage itself
  * Use a 16 bit depth buffer instead of a 24 bit one
  * Use a 16 bit color buffer instead of a 32 bit one
Bottleneck Identification

Now we have a bunch of practical ideas to find out if each stage is a bottleneck or not

Questions on Bottleneck Identification?
A Tool: NVPerfHud

- Free tool made to help identify bottlenecks
- Batches
- GPU idle
- CPU waits for GPU
- Driver time
- Total time
- Solid color pixel shaders
- 2x2 textures
- Etc...
Now let's look at some sample problems and see if we can find out where the problem is. Use NVPerfHUD to help.
Practice: Clean the Machine

- Make sure that your machine is ready for analysis
  - Make sure you have the right drivers
  - Use a release build of the game (optimizations on)
  - Check debug output for warnings or errors but.....
  - Use the **release** d3d runtime!!!
  - No maximum validation
  - No driver overridden anisotropic filtering or anti-aliasing
  - Make sure v-sync is off
Practice: Example 1

- A seemingly simple scene runs horribly slow
- Narrow in on the bottleneck

19.54 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra
Dynamic vertex buffer
BAD creation flags

HRESULT hr = pd3dDevice->CreateVertexBuffer(6 * sizeof(PARTICLE_VERT),0, //declares this as staticPARTICLE_VERT::FVF,D3DPOOL_DEFAULT,&m_pVB,NULL);
Practice: Example 1

- Dynamic vertex buffer
- GOOD creation flags

HRESULT hr = pd3dDevice->CreateVertexBuffer(
  6* sizeof(PARTICLE_VERT),
  D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY,
  PARTICLE_VERT::FVF,
  D3DPOOL_DEFAULT,
  &m_pVB, NULL );
Practice: Example 1

- Dynamic Vertex Buffer
  - BAD Lock flags

m_pVB->Lock(0, 0,(void**)quadTris, 0);

- No flags at all!?  
  - That can’t be good....
Practice: Example 1

Dynamic Vertex Buffer
  GOOD Lock flags

m_pVB->Lock(0, 0,(void**)&quadTris, D3DLOCK_NOSYSLOCK | D3DLOCK_DISCARD);

Use D3DLOCK_DISCARD the first time you lock a vertex buffer each frame
  And again when that buffer is full
  Otherwise just use NOSYSLOCK
Practice: Example 2

- Another slow scene
  - What's the problem here

9.55 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra
Practice: Example 2

Texture bandwidth overkill
- Use mipmaps
- Use dxt1 if possible
  - Some cards can store compressed data in cache
- Use smaller textures when they are fine
  - Does the grass blade really need a 1024x1024 texture?
    - Maybe
Practice: Example 3

- Another slow scene
- Who wants a prize?

9.41 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra
Practice: Example 3

- Expensive pixel shader
  - Can have huge performance effect
  - Only 3 verts, but maybe a million pixels
    - That’s only 1024x1024

Look at all the pixels!!
Practice: Example 3

36 cycles BAD

![Shader Perf output]

- Target: GeForceFX 5950 (NV38) : Unified Compiler: v56.58
- Cycles: 36 : # R Registers: 4
- GPU Utilization: 54.00%
- A large number of registers are being used which are causing register file stalls
- PS Instructions: 45
Practice: Example 3

11 cycles GOOD

Shader Perf

Target: GeForceFX 5950 (NV38) :: Unified Compiler: v56.58
Cycles: 11 :: # R Registers: 2
GPU Utilization: 54.00%
PS Instructions: 26
Practice: Example 3

What changed?
- Moved math that was constant across the triangle into the vertex shader
- Used ‘half’ instead of ‘float’
- Got rid of normalize where it wasn’t necessary
  - See Normalization Heuristics
Practice: Example 4

- The last one
- Audience: there are no more prizes, but we’ve locked the doors

47.26 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra
Practice: Example 4

- Too many batches
  - Was sending every quad as it’s own batch
  - Instead, group quads into one big VB then send that with one call

Number of DP calls: 2004
Practice: Example 4

What if they use different textures?

- Use texture atlases
- Put the two textures into a single texture and use a vertex and pixel shader to offset the texture coordinates
Balancing the Pipeline

Once satisfied with performance
- Balance the pipeline by making more use of un-bottlenecked stages
- Careful not to make too much use of them
Summary

- Pipeline architecture is ruled by bottlenecks
- Don’t waste time optimizing stages needlessly
- Identify bottlenecks with quick tests
- Use NVPerfHUD to analyze your pipeline
- Use Fxcomposer to help tune your shaders
- Check your performance early and often
  - Don’t wait until the last week!
Questions?

Ashu Rege (arege@nvidia.com)
Clint Brewer (cbrewer@nvidia.com)
Other NVIDIA programming talks

- GPU Gems Showcase
  Wed  5:30 – 6:00

- Real-time Translucent Animated Objects
  Fri  2:30 – 3:30
We collected some advice from various developers and include it here so you don’t have to discover it the hard way.
Performance Lore

- Use low resolution (<256x256) 8-bit normalization cube-maps. Quality isn’t reduced since 50% of texels in high resolution cube-map are identical you are only getting nearest filtering.
- Use oblique frustum clipping to clip geometry for reflection instead of a clip plane.
- Re-use vertex buffers for streaming geometry. Don’t create and delete vertex buffers every frame if they could be re-used.
- Use multiples of 32 byte sized vertices for transfer over AGP.
Performance Lore

- Use Occlusion Query and render object’s bounding box this frame. Then use the result next frame to decide whether or not you need to draw the real object.
- For ARB fragment programs use ARB_precision_hint_fastest.
- Use 16-bit 565 cube-maps for dynamic reflections on cars. Don’t need 32-bit reflections.
- Blend out small game objects and don’t render them when they are far away. Cuts down on batches.
Performance Lore

- use half instead of float optimizations early in development
- If rendering multiple passes, lay down Depth first then render your expensive pixel shaders. Cuts out depth complexity problems when shading
- If rendering multiple passes, on later additive passes you can set alpha to r + g + b, then use alpha test to cut on fill
- Terrain was rendered in 4 passes in ps1.1 due to texture limits. Render it in 1 pass in ps2.0
Performance Lore

- Communicate with IHVs about your problem, sometimes it really isn’t your code and we can fix the bugs!
- Use texture pages / atlases to combine objects into a single batch
- Use anisotropic filtering only on textures that need it. Don’t just set it to default on
- Don’t lock static vertex buffers multiple times per frame. make them dynamic
- Sorting the scene by render target gave a large perf boost
Performance Lore

- When locating the bottleneck, divide and conquer. Lower resolution first, cuts the problem almost in half. Rules out just about everything fill and pixel related.
- Use float4 to pack multiple float2 texture coordinates.
- Optimize your index and vertex buffers to take advantage of the cache.
- Move per object calculations out of the vertex shader and onto the CPU.
- Move per triangle calculations out of the pixel shader and into the vertex shader.
Performance Lore

- Use swizzles and masks in your vertex and pixel shaders: Value.xy = blah
- Use the API to clear the color and depth buffer
- Don’t change the direction of your z test mid frame, going from > ...to... >= ...to... = should be fine, but don’t go from > ...to... <
- Don’t use polygon offset if something else will work
- Don’t write depth in your pixel shader if you don’t have to
Use Mipmaps. If they are too blurry for you, use anisotropic and/or trilinear filtering: that gives better quality than LOD bias.

Rarely is there a single bottleneck in a game. If you find a bottleneck and fix it, and performance doesn’t improve more than a few fps. Don’t give up. You’ve helped yourself by making the real bottleneck apparent. Keep narrowing it down until you find it.
Bottleneck Identification

1. Run App
2. Vary FB
   - FPS varies? (Yes) FB limited
   - No
3. Vary texture size/filtering
   - FPS varies? (Yes) Texture limited
   - No
4. Vary resolution
   - FPS varies? (Yes) Vary fragment instructions
   - No
5. Vary vertex instructions
   - FPS varies? (Yes) Transform limited
   - No
6. Vary vertex size/AGP rate
   - FPS varies? (Yes) Transfer limited
   - No
7. Transform limited
8. Fragment limited
9. Raster limited
10. CPU limited

Yes

No
references


http://developer.nvidia.com/object/GDCE_2003_Presentations.html, Has other presentations on finding and locating the bottleneck
developer.nvidia.com
The Source for GPU Programming

- Latest documentation
- SDKs
- Cutting-edge tools
  - Performance analysis tools
  - Content creation tools
- Hundreds of effects
- Video presentations and tutorials
- Libraries and utilities
- News and newsletter archives

- Practical real-time graphics techniques from experts at leading corporations and universities

- Great value:
  - Contributions from industry experts
  - Full color (300+ diagrams and screenshots)
  - Hard cover
  - 816 pages
  - Available at GDC 2004


"GPU Gems is a cool toolbox of advanced graphics techniques. Novice programmers and graphics gurus alike will find the gems practical, intriguing, and useful."

Tim Sweeney
Lead programmer of Unreal at Epic Games

"This collection of articles is particularly impressive for its depth and breadth. The book includes product-oriented case studies, previously unpublished state-of-the-art research, comprehensive tutorials, and extensive code samples and demos throughout."

Eric Haines
Author of Real-Time Rendering