Ultimate Graphics Performance for DirectX 10 Hardware

Nicolas Thibieroz
European Developer Relations
AMD Graphics Products Group
nicolas.thibieroz@amd.com

V1.03
Generic API Usage

DX10 designed for performance

• No legacy code
• No fixed function
• Validation at creation time
• Immutable state objects
• User mode driver
• Powerful API
DX10 Batch Performance

The truth about DX10 batch performance

“Simple” porting job will not yield expected performance

Need to use DX10 features to yield gains:

• Geometry instancing
• Intelligent usage of state objects
• Intelligent usage of constant buffers
• Texture arrays
• Render Target selection (but...)
Geometry Instancing

Great instancing support in DX10

Best way to reduce batch overhead by far

Bind additional streams containing instance data

• And/or use “System Values” to vary rendering
  – SV_InstanceID, SV_PrimitiveID, SV_VertexID

Watch out for indexing too much data from CB!

• Additional vertex streams are usually preferable
State Management

DX10 uses immutable “state objects”

DX10 state objects require a new way to manage states

• A naïve DX9 to DX10 port will cause problems here
• Always create state objects at load-time
• Avoid duplicating state objects
• Recommendation to sort by states still valid in DX10!
• Implement “dirty states” mechanism to avoid redundancy
Constant Buffers Management

Major cause of slow performance in current DX10 apps!

Constants are declared in buffers in DX10

cbuffer PerFrameUpdateConstants
cbuffer SkinningMatricesConstants
{
    float4x4 mView;
    float fTime;
    float3 fWindForce;
    // etc.
}

When a CB has been updated and bound to a shader stage its whole contents are uploaded to the GPU

Need to strike a good balance between:

• Amount of constant data to upload
• Number of calls required to do it
Constant Buffers Management (2)

Always use a pool of constant buffers sorted by frequency of updates

- Don’t go overboard with number of CBs!
- Less than 5 is a good target
- CB sharing between shader stages can be a good thing

Global constant buffer unlikely to yield good performance
- Especially with regard to CB contention

Group constants by access patterns in a given buffer

```cpp
// GOOD

cbuffer PerFrameConstants
{
    float4 vLightVector;
    float4 vLightColor;
    float4 vOtherStuff[32];
};

// BAD

cbuffer PerFrameConstants
{
    float4 vLightVector;
    float4 vOtherStuff[32];
    float4 vLightColor;
};
```
Resource Updates

In-game creation and destruction of resources is slow!
• Runtime validation, driver checks, memory allocation…

Take into account for resource management
• Especially with regard to texture management

Create all resources in non-performance situations
• Up-front, level load, cutscenes, etc.

At run-time replace contents of existing resources rather than destroying/creating new ones
Resource Updates: Textures

Avoid `UpdateSubresource()` for texture updates
- Slow path in DX10 (think `DrawPrimitiveUP()` in DX9)
- Especially bad with larger textures!
- E.g. texture atlas, imposter pages, streaming data...

Perform all updates into pool of `D3D10_USAGE_STAGING` textures
- Use `Map(D3D10_MAP_WRITE, ...)` with `D3D10_MAP_FLAG_DO_NOT_WAIT` to avoid stalls

Then upload staging resources into video memory
- `CopyResource()`
- `CopySubresourceRegion()`
Resource Updates: Textures (2)

UpdateSubresource

UpdateSubresource

UpdateSubresource

D3D10_USAGE_DEFAULT

D3D10_USAGE_DEFAULT

D3D10_USAGE_STAGING

Map

CopySubresourceRegion

D3D10_USAGE_STAGING

Non-local Video Memory

Video Memory

11/08/2008

DirectX 10 Performance Tips
Resource Updates: Textures (3)

UpdateSubresource
UpdateSubresource
UpdateSubresource

D3D10_USAGE_DEFAULT
D3D10_USAGE_DEFAULT
D3D10_USAGE_DEFAULT

Map
Map
Map

CopySubresourceRegion
CopySubresourceRegion
CopySubresourceRegion

D3D10_USAGE_STAGING
D3D10_USAGE_STAGING
D3D10_USAGE_STAGING

Non-local Video Memory
Video Memory

11/08/2008 DirectX 10 Performance Tips
Resource Updates: Buffers

To update a Constant buffer
- `Map(D3D10_MAP_WRITE_DISCARD, ...);`
- `UpdateSubResource()`

To update a dynamic Vertex/Index buffer
- `Map(D3D10_MAP_WRITE_NO_OVERWRITE, ...);`
- Ring-buffer type; only write to empty portions of buffer
  - `Map(D3D10_MAP_DISCARD)` when buffer full
- `UpdateSubresource()` not as good as `Map()` in this case
Geometry Shader Recommendations

GS can write data out to memory
• This gives you “free” ALUs because of latency

Minimize the size of your output vertex structure
• Yields higher throughput
• Allows more output vertices (max output is 1024 floats)
• Consider adding pixel shader work to reduce output size

```c
[maxvertexcount(18)]
GSOUTPUT GS(point GSINPUT Input, inout PointStream<GSOUTPUT> OutputStream) {
    GSOUTPUT Output;
    for (int i=0; i<nNumPoints; i++) {
        // ...
        OutputStream.Append(Output);
    }
}

float4 PS(float4 Pos : SV_POSITION) : SV_TARGET
{
    float4 position = mul(float4(Pos, 1.0), mInvViewProjectionViewport);
    vLight = LightPos.xyz - Position.xyz;
    vView = CameraPos.xyz - Position.xyz;
}
```
Geometry Shader Recommendations 2

Cull triangles in Geometry Shader
• Backface culling
• Frustum culling

Minimize the size of your input vertex structure
• Combine inputs into single 4D iterator (e.g. 2 UV pairs)
• Pack data and use ALU instructions to unpack it
• Calculate values instead of storing them (e.g. binormal)
Color Clears

Clearing of color render targets is not free
• Cost is proportional to number of pixels to clear
• The less pixels to clear the better!

Here the rule about minimum work applies:
• Only clear render targets that need to be cleared!

Exception for MSAA RTs that need clearing every frame
RT clears are not required for optimal multi-GPU usage
Shader Model 4

Good writing practices enable optimal throughput:

• Write parallel code
• Avoid scalar dependencies

Execute scalar instructions first when mixed with vector ops

• Parentheses can help

Some instructions operate at a slower rate

• Integer multiplication and division
• Type conversions (float to int, int to float)
• Others (depends on IHV)

Declare constants in the right format
Shader Model 4 - continued

All DX10(.1) hardware support more maths instructions than texture operations
- Always target a high ALU:TEX ratio when writing shaders!
- Ratio affected by texture type, format and instruction

Use dynamic branching to skip instructions
- Can be a huge saving
- Especially when TEX instructions can be skipped
- But make sure branching has high coherency
- And don’t go overboard with branches!

Watch out for iterator bottleneck in pixel shader
- Always pack pixel shader inputs
Shader Compiler – Register Pressure

Indexed temporaries will reduce performance when stored in external video memory
• Avoid declaring arrays/structures of temporaries
• The performance impact of temp indexing can be dramatic!

Indexed constants may also increase register pressure
• Especially when indexing larger arrays

You can detect such cases with GPUShaderAnalyzer
• Watch out for the following instructions:
  MEM_SCRATCH_WRITE / MEM_SCRATCH_READ
  VFETCH
Alpha Test

Alpha test deprecated in DirectX 10
• Use `discard()` or `clip()` in HLSL pixel shaders
This requires the creation of two shader versions in DX10!
• One without `clip()` for opaque primitives
• One with `clip()` for transparent primitives

Don’t be tempted to use a single `clip()`-enabled shader
• This will impact GPU performance w.r.t. Z culling
• A single shader with a static/dynamic branch will still not be as good as having two versions

Side-effect: contribution towards “shaders explosion”

Put `clip()` / `discard()` as early as possible in shader
• Compiler may be able to skip remaining instructions
Accessing Depth and Stencil

DX10 enables the depth buffer to be read back as a texture

Enables features without requiring a separate depth render

• Atmosphere pass
• Soft particles
• DOF
• Forward shadow mapping
• Screen-space ambient occlusion

This is proving very popular to most game engines
Accessing Depth and Stencil with MSAA

DX10.0: reading a depth buffer as SRV is only supported in *single sample* mode
- Requires a separate depth render path for MSAA

Workarounds:
- Store depth in alpha of main FP16 RT
- Render depth into texture in a depth pre-pass
- Use a secondary render target in main color pass

DX10.1 allows depth buffer access as Shader Resource View in all cases
Shadow Maps

Avoid using `DXGI_FORMAT_D24_UNORM_S8_UINT` for depth shadow maps
• Reading back 24-bit format is a slow path on AMD GPUs
• No need for stencil for shadow maps anyway

Recommended depth shadow map formats:

- **`DXGI_FORMAT_D16_UNORM`**
  • Fastest shadow map format
  • Precision is enough in most situations
    – Just need to set your projection matrix optimally

- **`DXGI_FORMAT_D32_FLOAT`**
  • High-precision but slower than the 16-bit format
Multisampling Anti-Aliasing

Remember to NOT create your back buffer as multisampled!

- In most cases all multisampled rendering occurs on RT
  - Typically FP16 render target for HDR rendering
- Back buffer only receives resolved contents of RT

MSAA resolve operations are not free (on any hardware)
- This means `ResolveSubresource()` costs performance
- It is essential to limit the number of MSAA resolves
- Requires good design of effects and post-process chain
DX10.1 Overview

DX10.1 incremental update to DX10
• Released with Windows Vista SP1
• Supported on current graphic hardware

Adds new features and performance paths
• Fixed most DX10 “unorthogonalities”
  – Mandatory HW support (4xAA, FP32 filtering…)
  – Resource copies
  – Better support for MSAA
• Allow new algorithms to be implemented
  – Mandatory AA sample location
  – Shader Model 4.1
  – Etc.
DX10.1 Performance Advantages

Multi-pass reduction
• MSAA depth buffer readback as texture
• Per-sample PS execution & per-pixel coverage output mask
• 32 pixel shader inputs
• Individual MRT blend modes

More with less
• Gather4 (better/faster filter for SSAO or shadows)
• Cube map arrays for e.g. global illumination
• Etc.
Conclusion

Ensure the DX10 API is used optimally

- Sort by states
- Geometry Instancing
- Right balance of resource updates
- Right flags to Map() calls

Limit Geometry Shader output size

Write ALU-friendly shaders

Use DX10.1 if supported

Talk to IHVs!

11/08/2008

DirectX 10 Performance Tips
Questions?

nicolas.thibieroz@amd.com