GCN Performance Tweets

AMD Developer Relations

Overview

This document lists all GCN (“Graphics Core Next”) performance tweets that were released on Twitter during the first few months of 2013.
Each performance tweet in this document is accompanied by additional details to complete the information provided on Twitter.

GCN Performance Tip 1: Issues with Z-Fighting? Use D32_FLOAT_S8X24_UINT format with no performance or memory impact compared to D24S8.
Notes: Depth and stencil are stored separately on GCN architectures. The D32_FLOAT_S8X24_UINT is therefore not a 64-bit format like it could appear to be. There is no performance or memory footprint penalty from using a 32-bit depth buffer compared to using a 24-bit one.

GCN Performance Tip 2: Binding a depth buffer as a texture will decompress it, making subsequent Z ops more expensive.
Notes: After decompression the tiles of a depth buffer will stay in a decompressed state until they get completely covered or until the next Clear() operation on the whole buffer. A decompressed state means that Z operations will be slower than normal.

GCN Performance Tip 3: Invest in DirectCompute R&D to unlock new performance levels in your games.
Notes: DirectCompute, OpenCL and OpenGL compute shaders allow better access of shader features to the programmer.
Direct control over execution kernel, explicit thread synchronization and the use of shared memory are very powerful features that can unlock new performance levels in modern algorithms.
DirectCompute, OpenCL and OpenGL compute shaders should be equally fast in terms of raw compute performance (ignoring potential compiler efficiency differences).

GCN Performance Tip 4: On current GCN DX11 drivers the maximum recommended size for NO_OVERWRITE dynamic buffers is 4MB.
Notes: This applies to DYNAMIC buffers in general. The default “rename” memory size for DYNAMIC buffer on current GCN drivers is 8 MB, which means that contention on a 4MB buffer can be avoided at least once. If a DYNAMIC buffer is larger than 4 MB then

a Map() DISCARD operation on this buffer will introduce a synchronization between CPU and GPU which is bad news for performance.

GCN Performance Tip 5: Limit Vertex and Domain Shader output size to 4 float4/int4 attributes for best performance.
Notes: Outputs larger than 4 float4/int4 have increased parameter cache storage requirements which reduce wave occupancy (the number of wavefronts “in flight”) and may therefore impact performance. As an added bonus using fewer DS outputs will reduce PS interpolation cost.

GCN Performance Tip 6: RGBA16 and RGBA16F are fast export, use those to pack G-Buffer data and avoid ROP bottlenecks.
Notes: With Deferred renderers it may be faster to pack G-Buffer properties into RGBA16(F) render targets in order to reduce the total export cost of the G-Buffer building phase. E.g. 4x RGBA8888 render targets is 4 cycles export whereas 2x RGBA16(F) is 2 cycles.

GCN Performance Tip 7: Design your game engine with geometry instancing support from an early stage.
Notes: Geometry instancing is essential to reduce the total number of draw calls in PC engines.

GCN Performance Tip 8: Pure Vertex Shader-based solutions can be faster than using the GS or HS/DS.
Notes: In some cases the Vertex Shader can be used to completely replace fixed-expansion cases that the Geometry Shader or the tessellation shaders would normally be responsible for. Small amplification and/or short shaders are often good candidates for this optimization.
A pure Vertex Shader-based solution can be implemented with Vertex Shader “instancing”: this consists in rendering additional vertices at the Draw call level and using System Values like SV_InstanceID and SV_VertexID to control their outputs. For example front-facing particles (4 vertices) can easily be done with a Vertex Shader instancing solution. Particle lighting or shadowing performed at vertex frequency can also be done this way instead of resorting to enabling Hull and Domain shaders, etc.

GCN Performance Tip 9: Use a ring of STAGING resources to update textures. UpdateSubresource is slow unless texture size is <4Kb.
Notes: It is more efficient to perform all updates into a STAGING textures first, and then upload the STAGING texture into a DEFAULT texture via a CopyResource() operation. The UpdateSubresource() path is generally not optimal for texture updates due to the overhead of additional copies occurring at the runtime and driver level.

GCN Performance Tip 10: DX11 supports free-threaded resource creation, use it to reduce shader compilation and texture loading times.
Notes: With shaders becoming longer and more complex it is important to try to minimize compilation time by spreading this cost onto multiple threads.

GCN Performance Tip 11: Use the smallest Input Layout necessary for a given VS; this is especially important for depth-only rendering.

Notes: Vertex structures often contain a variety of inputs but only a small selection of those are required for depth-only rendering (position and texture coordinates for alpha-tested geometry). Only binding the required inputs in a separate vertex buffer will result in better cache utilization and therefore better performance.

GCN Performance Tip 12: Don't forget to optimize geometry for index locality and sequential read access - including procedural geometry.
Notes: Index re-use is important to minimize Vertex Shader execution cost, especially in depth-only rendering situations where the GPU front-end is more likely to be a bottleneck.

GCN Performance Tip 13: Implement backface culling in Hull Shader if tessellation factors are on the high side.
Notes: Fixed-function backface culling occurs at the rasterization stage which is *after* the tessellation stages. Tessellating a primitive has a cost that is directly associated with the tessellation factors computed. Implementing conservative backface culling in the Hull Shader is an efficient way to limit the cost of tessellation by ensuring that only visible (front-facing) primitives are tessellated.
Input primitives can be culled in the Hull Shader by setting one of their tessellation factors to 0.

GCN Performance Tip 14: Use flow control in shaders but watch out for GPR pressure caused by deep nested branches.
Notes: Flow control is very efficient on GCN architectures and should be used to write fast shaders. However deep nesting of branches is likely to increase the General Purpose Registers requirement for this shader which in turn would reduce wave occupancy (number of wavefronts scheduled per SIMD), affecting performance.

GCN Performance Tip 15: Some shader instructions are costly; pre-compute constants and store them in CB (e.g. reciprocals).
Notes: Some shader instructions are more costly than others: SIN, COS, RCP, RSQ, integer MUL and DIV etc. To avoid using those instructions it is recommended to precompute any constant value requiring their use into constant buffers.

GCN Performance Tip 16: Use [maxtessfactor(X)] in Hull Shader declaration to control tessellation costs. Max recommended value is 15.
Notes: High tessellation factors can have a major impact on overall performance. Tessellation performance will decrease fairly linearly with increasing tessellation factors up to a value of 15 after which performance falls off a cliff. The use of [maxtessfactor(15)] can be used to keep those costs under controls.

GCN Performance Tip 17: Filtering 64-bit texture formats is half-rate on current GCN architectures, only use if needed.
Notes: Bilinear filtering on RGBA16(F), RG32(F) take two cycles to execute (compared to a single cycle for 32-bit formats).

GCN Performance Tip 18: clip, discard, alpha-to-mask and writing to oMask or oDepth disable Early-Z when depth writes are on.

Notes: Early-Z is the per-pixel depth rejection test that occurs before the pixel shader stage in the GPU. For performance reasons it should be enabled as often as possible. If depth writes are disabled then doing clip, discard, alpha-to-mask and writing to oMask will not disable Early-Z.

GCN Performance Tip 19: Writing to a UAV or oDepth disables both Early-Z and Hi-Z unless conservative oDepth is used.
Notes: Declaring [earlydepthstencil] in front of a pixel shader guarantees the depth test happens early even if the shader writes to a UAV.

GCN Performance Tip 20: DispatchIndirect() and Draw[Indexed]InstancedIndirect() can be used to implement some form of conditional rendering.
Notes: These DirectX11 API functions take their input parameters (number of primitives or threadgroups to execute) from buffers that can be written to in previous draw/compute operations.

GCN Performance Tip 21: Use a multiple of 64 in Compute Shader threadgroup declaration. 256 is often a good choice.
Notes: 64 is the wavefront size on current GCN-based GPUs. Using a threadgroup declaration size that’s not a multiple of 64 will therefore result in wasted processing. 256 or less is often a good choice because it may allow up to 4 or more wavefronts-worth of work to be scheduled on a compute unit for latency hiding (depending on GPR and Thread Group Shared Memory availability).

GCN Performance Tip 22: Occlusion queries will stall the CPU if not used correctly.
Notes: DirectX11 occlusion queries results are provided to the CPU. Application requesting immediate occlusion query results will introduce a stall because the CPU will no longer be able to send new command buffer data to the GPU. Instead occlusion queries must be given enough time to complete before their results are requested. Results should ideally be queried when made available, or at the very least a number of frames should be allowed to execute between the initiation of the query and getting its result back. This number of frames should be at least equal to the number of GPUs in the system to allow parallelism between CPU and GPU(s).
In OpenGL the AMD_query_buffer_object extension can be used to copy the result of an occlusion query into buffer object memory in order to avoid a round trip to the CPU.
It is also important to keep the number of occlusion queries “in-flight” under control. No more than a couple of hundreds occlusion queries per frame should be used.

GCN Performance Tip 23: GetDimensions() is a TEX instruction; prefer storing texture dimensions in a Constant Buffer if TEX-bound.
Notes: The use of a TEX instruction can lead to lower shader performance if the shader is TEX instructions-limited. It may also be subject to additional latency. It is therefore preferred to store the desired texture dimensions into a constant buffer.

GCN Performance Tip 24: Avoid indexing into arrays of shader variables - this has a high performance impact.

Notes: If indexing cannot be resolved at compile time then indexing into arrays of shader variables will cause these to be stored in Vector Generic Purpose Registers or scratch memory. In either case the performance impact of such allocation is significant, and likely to impact latency hiding.

GCN Performance Tip 25: Pack Vertex Shader outputs to a float4 vector to optimize attributes storage.
Notes: It is more efficient to ensure all Vertex Shader outputs are packed into float4 vectors to ensure optimal utilization, especially if it allows four or less float4 vector outputs to be used (see tip #5). While some form of packing will be attempted by the HLSL compiler it will typically not be able to do as good as job as the programmer. For example exporting two float3 and one float2 should be packed into two float4 vectors instead.

GCN Performance Tip 26: Coalesce reads and writes to Unordered Access Views to help with memory accesses.
Notes: This is true of all memory accesses in general. As all threads of a wavefront are executed in parallel it is a good idea to arrange memory access in a way to optimize memory cache efficiency across all threads. For the same reason it is recommended to prefer Structure of Arrays (SoA) over Arrays of Structures (AoS) to optimize cache access patterns in parallel computing environments.

GCN Performance Tip 27: Render your skybox last and your first-person geometry first (should be a given by today's standards :)).
Notes: The DirectX and OpenGL graphics pipelines place the pixel shader stage before the depth test. However on the GPU the depth test will be executed first whenever possible to avoid shading pixels that end up being culled by depth testing. Rendering close-up geometry first is a good way to ensure the depth buffer is primed with small values to maximize the potential for fragment rejection via Hierarchical Z and Early-Z testing. Thus for a first or third person game rendering the playable character first is more efficient than rendering it last. In the same vein geometry that is distant like a skybox should be rendered last to ensure that as many fragments as possible are depth-rejected prior to shading.

GCN Performance Tip 28: Using D16 shadow maps will provide a modest performance boost and a large memory saving.
Notes: D16 shadow maps are half the memory footprint of D24X8 or D32 shadow maps. D24 shadow maps are internally stored in a 32-bit format (as explained in tip #1). D16 will run slightly faster than other depth-only formats because of reduced memory bandwidth. Concerns regarding depth precision of D16 are usually unfounded: in most cases pushing the front clip plane as much as possible results in a much better depth distribution and therefore avoid precision issues.

GCN Performance Tip 29: Avoid unnecessary DISCARD when Map()ping resources; some apps still do this at least once a frame.
Notes: A Map() with DISCARD operation may trigger a driver-side "rename" operation whereby the buffer being DISCARD-ed is given a fresh pointer in a new memory allocation if its previous contents are still being used by the GPU. To avoid renaming overhead (and running out of rename memory – see tip #4) it is therefore important to

reduce usage of DISCARD in buffer updates. There is no need to DISCARD a buffer once a frame; instead DYNAMIC buffers used with NO_OVERWRITE updates should only be DISCARD-ed when full.

GCN Performance Tip 30: Minimize GS input and output size or consider Vertex Shader-instancing solutions instead.
Notes: The Geometry Shader typically has a large performance overhead in case of large and/or variable amplification. To minimize the performance impact of this shader stage it is important to reduce the size of both input and output data.
For fixed amplification it is generally better to prefer a Vertex Shader-instancing solution instead whereby additional vertices are specified at the Draw call level and system values such as SV_VertexID and SV_InstanceID used to control their placement (as explained in tip #8).

GCN Performance Tip 31: A dedicated thread solely responsible for making D3D calls is usually the best way to drive the API.
Notes: The best way to drive a high number of draw calls in DirectX11 is to dedicate a thread to graphics API calls. This thread's sole responsibility should be to make DirectX calls; any other types of work should be moved onto other threads (including processing memory buffer contents). This graphics "producer thread" approach allows the feeding of the driver's "consumer thread" as fast as possible, enabling a high number of API calls to be processed.
DirectX11 Deferred Contexts will not achieve faster results than this approach when it is implemented correctly.

GCN Performance Tip 32: Avoid sparse shader resource slot assignments, e.g. binding resource slot #0 and #127 is a bad idea.
Notes: Resource descriptors are stored as an array in graphics memory and managing these descriptors is subject to additional efficiency and memory overhead when a large range of resource slots is used.

GCN Performance Tip 33: Thread Group Shared Memory accesses are subject to bank conflicts on addresses that are multiples of 32 DWORD.
Notes: Having multiple threads read from the *same* TGSM address is not subject to such bank conflicts.

GCN Performance Tip 34: The D3DXSHADER_IEEE_STRICTNESS shader compiler flag is likely to produce longer shader code.
Notes: This flag enforces additional precision on certain ALU operations, leading to more/more costly instructions being used.

GCN Performance Tip 35: Use D3D11_USAGE_IMMUTABLE on read-only resources. A surprising number of games don't!
Notes: The more information is provided to the drivers and the runtime the better. Games and application often include resources that are never updated and those should be created with the IMMUTABLE flag to optimize memory management. For example skybox and HUD textures are likely to qualify for this flag.

GCN Performance Tip 36: Avoid calling Map() on DYNAMIC textures as this may require a conversion from tiled to linear memory.
Notes: A DYNAMIC texture may be stored in a non-linear hardware-friendly format. Map()ping such a texture for read or write operations may therefore force the conversion of this data into a linear format which introduces a performance overhead.

GCN Performance Tip 37: Only use UAV BIND flag when needed. Leaving it on for non-UAV usage will cause perf issues due to extra sync.
Notes: GCN graphics drivers perform surface synchronization tasks to ensure a surface that's been written to is ready to be used again in subsequent graphics operations. The cost of this synchronization will be larger if the BIND flags provided indicate both D3D11_BIND_UNORDERED_ACCESS and D3D11_BIND_RENDER_TARGET.

GCN Performance Tip 38: Passing interpolated screenpos can be better than declaring SV_POSITION in pixel shader especially if PS is short.
Notes: Declaring SV_POSITION in the pixel shader will not be as efficient as passing screen coordinates from the previous shader stage because its use is hard-coded to fixed-function hardware that is also used for other purposes.
If a pixel shader needs access to fragment position it is recommended to pass it via texture coordinates instead.

GCN Performance Tip 39: Ensure proxy and predicated geometry are spaced by a few draws when using predicated rendering.
Notes: This is to ensure that proxy geometry has finished drawing *and* its occlusion results have made it to memory before the decision of drawing predicated geometry is made. Not leaving enough draw operations between the two can result in predicated geometry being drawn regardless of occlusion results, defeating the point of predication in the first place.

GCN Performance Tip 40: Fetch indirections increase execution latency; keep it under control especially for VS and DS stages.
Notes: A "fetch indirection" refers to the process of fetching memory data whose address is itself depending on a previous memory fetch operation. Such memory fetches cannot be grouped together since one depends on the other. Because of the latency involved in fetching memory such dependencies will therefore increase total execution latency.

GCN Performance Tip 41: Dynamic indexing into a Constant Buffer counts as fetch indirection and should be avoided.
Notes: If a calculated index is different across all threads of a wavefront then the fetch of Constant Buffer data using such index is akin to a memory fetch operation.

GCN Performance Tip 42: Always clear MSAA render targets before rendering.
Notes: Multisampled render targets are stored in compressed format to improve performance. Not clearing MSAA render targets over time would accumulate a wide range of pixel color data into the multisampled buffer to the point where compression is no longer able to represent this data accurately, resulting in affected tiles turning into a decompressed state. This decompressed state is bad news for performance, and can

be avoided by ensuring multisampled render targets are Clear()ed to reset compression whenever their contents are no longer needed.

GCN Performance Tip 43: With cascaded shadow maps use area culling to exclude geometry already rendered in finer shadow cascades.
Notes: With cascaded shadow maps geometry that is rendered in a previous (finer) cascade typically need not be rendered again in coarser cascades. It can therefore be beneficial to set up a culling scheme on the CPU side to only send geometry that will be used in those coarser cascades. This is especially important as depth-only rendering is usually front-end limited thus rendering less geometry in cascaded shadow maps will directly benefit performance.

GCN Performance Tip 44: Avoid over-tessellating geometry that produces small triangles in screen space; in general avoid tiny triangles.
Notes: The smallest work unit in modern GPUs is the pixel quad (2x2 pixels). Small triangles have efficiency problems because fitting 2x2 pixel quads to cover their area is very likely to produce poor quad occupancy. Poor quad occupancy results in a waste of GPU resources and should therefore be avoided by adopting suitable LOD systems for geometry, especially when tessellation is used.

GCN Performance Tip 45: Create shaders before textures to give the driver enough time to convert the D3D ASM to GCN ASM.
Notes: GCN drivers defer compilation of shaders onto separate threads. Creating shaders early on during the loading process ensures they have enough time to finish compiling before the game starts. To ensure all shaders have finished compiling always warm the shader cache by binding all needed shaders into an offscreen rendering operation prior to rendering the game level.

GCN Performance Tip 46: Atomic operations on a single TGSM location from all threads will serialize TGSM access.
Notes: Atomic operations ensure that the read/modify/write operation specified will be carried out without being interrupted by other operations on the same memory address. If multiple threads attempts to perform an atomic operation (called Interlocked*() in Shader Model 5.0) onto the same Thread Group Shared Memory address then those operations therefore have to be serialized to ensure atomicity. This serialization is likely to have a considerable impact on performance, and should be avoided.

GCN Performance Tip 47: Improve motion blur performance by POINT sampling along motion vectors if your source image is RGBA16F.
Notes: Bilinear fetches into 64-bit textures are half-rate on GCN architectures. Motion blur effects often sample pixels along a line along motion vectors so if the source texture is a 64-bit format the effect is likely to be bottlenecked by texture fetch operations. The use of POINT sampling for fetching pixels is not only likely to improve performance of this effect but also it will not produce a discernible visual difference compared to bilinear filtering due to the low-frequency nature of motion blur.
Another alternative is to use a 32-bits input texture for the motion blur effect (point sampling and bilinear fetches into 32-bit textures are full-rate).

GCN Performance Tip 48: MIPMapping is underrated - don't forget to use it on displacement maps and volume textures too.
Notes: The use of MIPMapping is essential to avoid aliasing issues and improve texture cache performance. It should especially be used on volume textures as they are more likely to have poor cache hit rates. Non-color data such as normal maps and displacement maps should also use MIPMapping.

GCN Performance Tip 49: Trilinear is up to 2x the cost of bilinear. Bilinear on 3D textures is 2x the cost of 2D. Aniso cost depends on taps
Notes: Trilinear filtering performs a bilinear operation on the two nearest MIP levels over the transition range.
Two bilinear fetches are required when sampling from a volume texture with bilinear filtering.
The cost of anisotropic filtering depends on the number of taps applied. The more taps the more fetches will be performed. Note that the number of taps depends on the anisotropic filtering quality specified (2x, 4x, 8x or 16x), and the number of taps derived from the texture coordinate derivatives.

GCN Performance Tip 50: Avoid heavy switching between compute and rendering jobs. Jobs of the same type should be done consecutively.
Notes: GCN drivers have to perform surface synchronization tasks when switching between compute and rendering tasks. Heavy back-and-forth switching may therefore increase synchronization overhead and reduce performance.

Revision history:

23 May 2013: First revision.