The ryg blog
When I grow up I'll be an inventor.

A trip through the Graphics Pipeline 2011, part 3

July 3, 2011
This post is part of the series ***"A trip through the Graphics Pipeline 2011" (https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/)****.*

At this point, we've sent draw calls down from our app all the way through various driver layers and the command processor; now, *finally* we're actually going to do some graphics processing on it! In this part, I'll look at the vertex pipeline. But before we start…

Have some Alphabet Soup!

We're now in the 3D pipeline proper, which in turn consists of several stages, each of which does one particular job. I'm gonna give names to all the stages I'll talk about – mostly sticking with the "official" D3D10/11 names for consistency – plus the corresponding acronyms. We'll see all of these eventually on our grand tour, but it'll take a while (and several more parts) until we see most of them – seriously, I made a small outline of the ground I want to cover, and this series will keep me busy for at least 2 weeks! Anyway, here goes, together with a one-sentence summary of what each stage does.

`IA` — Input Assembler. Reads index and vertex data.
`VS` — Vertex shader. Gets input vertex data, writes out processed vertex data for the next stage.
`PA` — Primitive Assembly. Reads the vertices that make up a primitive and passes them on.
`HS` — Hull shader; accepts patch primitives, writes transformed (or not) patch control points, inputs for the domain shader, plus some extra data that drives tessellation.
`TS` — Tessellator stage. Creates vertices and connectivity for tessellated lines or triangles.
`DS` — Domain shader; takes shaded control points, extra data from HS and tessellated positions from TS and turns them into vertices again.
`GS` — Geometry shader; inputs primitives, optionally with adjacency information, then outputs different primitives. Also the primary hub for…
`SO` — Stream-out. Writes GS output (i.e. transformed primitives) to a buffer in memory.
`RS` — Rasterizer. Rasterizes primitives.
`PS` — Pixel shader. Gets interpolated vertex data, outputs pixel colors. Can also write to UAVs (unordered access views).
`OM` — Output merger. Gets shaded pixels from PS, does alpha blending and writes them back to the backbuffer.
`CS` — Compute shader. In its own pipeline all by itself. Only input is constant buffers+thread ID; can write to buffers and UAVs.

And now that that's out of the way, here's a list of the various data paths I'll be talking about, in order: (I'll leave out the IA, PA, RS and OM stages in here, since for our purposes they don't actually do anything *to* the data, they just rearrange/reorder it – i.e. they're essentially glue)

1. VS→PS: Ye Olde Programmable Pipeline. In D3D9, this was all you got. Still the most important path for regular rendering by far. I'll go through this from beginning to end then double back to the fancier paths once I'm done.
2. VS→GS→PS: Geometry Shading (new with D3D10).
3. VS→HS→TS→DS→PS, VS→HS→TS→DS→GS→PS: Tessellation (new in D3D11).
4. VS→SO, VS→GS→SO, VS→HS→TS→DS→GS→SO: Stream-out (with and without tessellation).
5. CS: Compute. New in D3D11.

And now that you know what's coming up, let's get started on vertex shaders!

Input Assembler stage

The very first thing that happens here is loading indices from the index buffer – if it's an indexed batch. If not, just pretend it was an identity index buffer (0 1 2 3 4 …) and use that as index instead. If there is an index buffer, its contents are read from memory at this point – not directly though, the IA usually has a data cache to exploit locality of index/vertex buffer access. Also note that index buffer reads (in fact, all resource accesses in D3D10+) are bounds checked; if you reference elements outside the original index buffer (for example, issue a `DrawIndexed` with `IndexCount == 6` from a 5-index buffer) all out-of-bounds reads return zero. Which (in this particular case) is completely useless, but well-defined. Similarly, you can issue a `DrawIndexed` with a `NULL` index buffer set – this behaves the same way as if you had an index buffer of size zero set, i.e. all reads are out-of-bounds and hence return zero. With D3D10+, you have to work some more to get into the realm of undefined behavior. :)

Once we have the index, we have all we need to read both per-vertex and per-instance data (the current instance ID is just another counter, fairly straightforward, at this stage anyway) from the input vertex streams. This is fairly straightforward – we have a declaration of the data layout; just read it from the cache/memory and unpack it into the float format that our shader cores want for input. However, this read isn't done immediately; the hardware is running a cache of shaded vertices, so that if one vertex is referenced by multiple triangles (and in a fully regular closed triangle mesh, each vertex will be referenced by about 6 tris!) it doesn't need to be shaded every time – we just reference the shaded data that's already there!

Vertex Caching and Shading

Note: The contents of this section are, in part, guesswork. They're based on public comments made by people "in the know" about current GPUs, but that only gives me the "what", not the "why", so there's some extrapolation here. Also, I'm simply guessing some of the details here. That said, I'm not talking completely out of my ass here – I'm confident that what I'm describing here is both reasonable and works (in the general sense), I just can't guarantee that it's actually that way in real HW or that I didn't miss any tricky details. :)

Anyway. For a long time (up to and including the shader model 3.0 generation of GPUs), vertex and pixel shaders were implemented with different units that had different performance trade-offs, and vertex caches were a fairly simple affair: usually just a FIFO for a small number (think one or two dozen) of vertices, with enough space for a worst-case number of output attributes, using the vertex index as a tag. As said, fairly straightforward stuff.

And then unified shaders happened. If you unify two types of shaders that used to be different, the design is necessarily going to be a compromise. So on the one hand, you have vertex shaders, which (at that time) touched maybe up to 1 million vertices a frame in normal use. On the other hand you had pixel shaders, which at 1920×1200 need to touch *at least* 2.3 million pixels a frame *just to fill the whole screen once* – and a lot more if you want to render anything interesting. So guess which of the two units ended up pulling the short straw?

Okay, so here's the deal: instead of the vertex shader units of old that shaded more or less one vertex at a time, you now have a huge beast of a unified shader unit that's designed for maximum throughput, not latency, and hence wants large batches of work (How large? Right now, the magic number seems to be between 16 and 64 vertices shaded in one batch).

So you need between 16-64 vertex cache misses until you can dispatch one vertex shading load, if you don't want to shade inefficiently. But the whole FIFO thing doesn't really play ball with this idea of batching up vertex cache misses and shading them in one go. The problem is this: if you shade a whole batch of vertices at once, that means you can only actually start assembling triangles once all those vertices have finished shading. At which point you've just added a whole batch (let's just say 32 here and in the following) of vertices to the end of the FIFO, which means 32 old vertices now fell out – but each of these 32 vertices might've been a vertex cache hit for one of the triangles in the current batch we're trying to assemble! Uh oh, that doesn't work. Clearly, we can't actually count the 32 oldest verts in the FIFO as vertex cache hits, because by the time we want to reference them they'll be gone! Also, how big do we want to make this FIFO? If we're shading 32 verts in a batch, it needs to be at least 32 entries large, but since we can't use the 32 oldest entries (since we'll be shifting them out), that means we'll effectively start with an empty FIFO on every batch. So, make it bigger, say 64 entries? That's pretty big. And note that every vertex cache lookup involves comparing the tag (vertex index) against all tags in the FIFO – this is fully parallel, but it also a power hog; we're effectively implementing a fully associative cache here. Also, what do we do between dispatching a shading load of 32 vertices and receiving results – just wait? This shading will take a few hundred cycles, waiting seems like a stupid idea! Maybe have two shading loads in flight, in parallel? But now our FIFO needs to be at least 64 entries long, and we can't count the last 64 entries as vertex cache hits, since they'll be shifted out by the time we receive results. Also, one FIFO vs. lots of shader cores? **Amdahl's law (http://en.wikipedia.org/wiki/Amdahl%27s_law)** still holds – putting one strictly serial component in a pipeline that's otherwise completely parallel is a surefire way to make it the bottleneck.

This whole FIFO thing really doesn't adapt well to this environment, so, well, just throw it out. Back to the drawing board. What do we actually want to do? Get a decently-sized batch of vertices to shade, and not shade vertices (much) more often than necessary.

So, well, keep it simple: Reserve enough buffer space for 32 vertices (=1 batch), and similarly cache tag space for 32 entries. Start with an empty "cache", i.e. all entries invalid. For every primitive in the index buffer, do a lookup on all the indices; if it's a hit in the cache, fine. If it's a miss, allocate a slot in the current batch and add the new index to the cache tag array. Once we don't have enough space left to add a new primitive anymore, dispatch the whole batch for vertex shading, save the cache tag array (i.e. the 32 indices of the vertices we just shaded), and start setting up the next batch, again from an empty cache – ensuring that the batches are completely independent.

Each batch will keep a shader unit busy for some while (probably at least a few hundred cycles!). But that's no problem, because we got plenty of them – just pick a different unit to execute each batch! Presto parallelism. We'll eventually get the results back. At which point we can use the saved cache tags and the original index buffer data to assemble primitives to be sent down the pipeline (this is what "primitive assembly" does, which I'll cover in the later part).

By the way, when I say "get the results back", what does that mean? Where do they end up? There's two major choices: 1. specialized buffers or 2. some general cache/scratchpad memory. It used to be 1), with a fixed organization designed around vertex data (with space for 16 float4 vectors of attributes per vertex and so on), but lately GPUs seem to be moving towards 2), i.e. "just memory". It's more flexible, and has the distinct advantage that you can use this memory for other shader stages, whereas things like specialized vertex caches are fairly useless for the pixel shading or compute pipeline, to give just one example.

Update: And here's a **picture (http://www.farbrausch.de/~fg/gpu/vertex_shade.jpg)** of the vertex shading dataflow as described so far.

Shader Unit internals

Short versions: It's pretty much what you'd expect from looking at disassembled HLSL compiler output (`fxc /dumpbin` is your friend!). Guess what, it's just processors that are *really good* at running that kind of code, and the way that kind of thing is done in hardware is building something that eats something fairly close to shader bytecode, in spirit anyway. And unlike the stuff that I've been talking about so far, it's fairly well documented too – if you're interested, just check out conference presentations from AMD and NVidia or read the documentation for the CUDA/Stream SDKs.

Anyway, here's the executive summary: fast ALU mostly built around a FMAC (Floating Multiply-ACcumulate) unit, some HW support for (at least) reciprocal, reciprocal square root, log2, exp2, sin, cos, optimized for high throughput and high density not low latency, running a high number of threads to cover said latency, fairly small number of registers per thread (since you're running so many of them!), very good at executing straight-line code, bad at branches (especially if they're not coherent).

All that is common to pretty much all implementations. There's some differences, too; AMD hardware used to stick directly with the 4-wide SIMD implied by the HLSL/GLSL and shader bytecode (even though they seem to be moving away from that lately), while NVidia decided to rather turn the 4-way SIMD into scalar instructions a while back. Again though, all that's on the Web already!

What's interesting to note though is the *differences* between the various shader stages. The short version is that really are rather few of them; for example, all the arithmetic and logic instructions are exactly the same across all stages. Some constructs (like derivative instructions and interpolated attributes in pixel shaders) only exist in some stages; but mostly, the differences are just what kind (and format) of data are passed in and out.

There's one special bit related to shaders though that's a big enough subject to deserve a part on its own. That bit is texture sampling (and texture units). Which, it turns out, will be our topic next time! See you then.

Closing remarks

Again, I repeat my disclaimer from the "Vertex Caching and Shading" section: Part of that is conjecture on my part, so take it with a grain of salt. Or maybe a pound. I don't know.

I'm also not going into any detail on how scratch/cache memory is managed; the buffer sizes depend (primarily) on the size of batches you process and the number of vertex output attributes you expect. Buffer sizing and management is *really* important for performance, but I can't meaningfully explain it here, nor do I want to; while interesting, this stuff is very specific to whatever hardware you're talking about, and not really very insightful.

From → Coding, Graphics Pipeline

13 Comments

1. **Won Chun permalink**
   Keep it up! These articles are great. I used to know this nitty gritty way better years ago, but things have changed so much since then and this has been a great way to catch up.

   So I thought one of the reasons why the vertex caches changed was not just because of the massive parallelism of the shaders, but also because primitive assembly became a bottleneck, and 1 tri/clock was no long sufficient. I could be wrong or redundant here, of course.

   Transparent post-transform vertex caching is pretty cool, but I always thought that it would have been a big win to dispose of the associative lookups, and use an explicit style (kind of like an extension to the old generalized strips approach). Without associative lookups, you could have much larger, explicitly indexed caches.

   The way it would work, is that instead of having index buffers, you'd have a list of back-references to previously seen vertices. These references could be small since they index the cache, rather than the vertex buffer; they could be 8-bit, instead of 16 or 32-bit. You would need a few escape codes, like to index a never-seen vertex, a previously-seen vertex out of the cache, and maybe a primitive restart code.

   Maybe this makes more sense for mobile (many implementations don't really have post-transform caches and rely only on strips, maybe because of the power issue) than desktop.

   Reply

   - **fgiesen permalink**
     "So I thought one of the reasons why the vertex caches changed was not just because of the massive parallelism of the shaders, but also because primitive assembly became a bottleneck, and 1 tri/clock was no long sufficient. I could be wrong or redundant here, of course."
     Yep, that's what I was hinting with the "the FIFO is a serial part of the pipeline" bit; basically PA needs to have the vertex data for all of the primitive somewhere, and if that's place is your FIFO, then you can have only one PA (because there's just one FIFO)! You could have multiple FIFOs in theory, but then you'd end up re-shading vertices for every FIFO individually, which is totally beside the point for a vertex cache. So yeah, just block it and dispatch/work on a chunk basis downstream. Way easier (and more scalable!) overall.

     "Transparent post-transform vertex caching is pretty cool, but I always thought that it would have been a big win to dispose of the associative lookups, and use an explicit style"
     So basically a two-level indexing scheme; I'm not sure. It's certainly a more efficient way to drive the pipeline in general, but interface compatibility is a big deal. No matter what they do, they'll still need to support regular indexed primitives efficiently, since that's what everyone uses up until now. At this point there's just significant friction involved in changing this; we might get there eventually if it becomes a significant issue, but right now that doesn't seem to be the case.

     Also note that e.g. with a 16-wide shader unit, you want to shoot for having batches at least as large as 16, but you can obviously shoot for wider multiples too: 32, 48, 64 and so on. This gives you better hit rates (this whole setup thing I described restarts from scratch in every block!) but also needs more memory for buffering and has higher granularity. If you go for a fully associative cache, there'll be a fixed limit of how large your maximum vertex shade batch can be (which is the size of that cache). But the better way to make it scale really wide is to just not be fully associative, and make the cache set-associative instead; now every vertex index has a couple places it can go in the cache. If you encounter an index that would go into a set that's already full, you can't add it to the current block anymore, so you have to dispatch what might be a partial block. Note that for triangles your cache needs to be at least 3-way associative so you can make progress – all vertex indices in a tri might map to the same set!

I don't think anyone currently does this though; for one, very wide batches need lots of buffering for output, and buffer space is limited, so larger batches mean you can run fewer of them in parallel. If you're running too few in parallel, you can't cover the latency and end up waiting. More importantly, I'm talking about D3D11 here; the worst-case primitive that needs to be supported is a 32-control point patch, which absolutely positively needs at least a 32-vertex batch in order to assemble a single primitive! (You need to be able to assemble at least one primitive at all times, or you could get stuck indefinitely). Which would also mean you need at least a 32-way associative cache in the set associative model!

"Maybe this makes more sense for mobile (many implementations don't really have post-transform caches and rely only on strips, maybe because of the power issue) than desktop."
The thing with strips is that you basically get them for free. In PA, you need buffers for at least 3 vertices (a single triangle) no matter what you do. But once you buffer the last 3 vertices, you get triangle strips effectively for free. A proper post-transform vertex cache, on the other hand, takes up area, power and design effort. If you don't care much about triangle throughput, it's not worth it.

Reply

- **zeuxcg permalink**
  As far as I know, the scheme more or less like yours was used in old GeForces ("old" means before GFFX, I think – maybe even before GF3). I also vaguely remember that dynamic index buffers were slow, which is related, of course.

  Still, the current way is better – it opens ways to do things differently (i.e. it's hard to do backreferences in a parallel way, the same way as it's hard to do traditional FIFO cache), it's conceptually simple and I don't think it wastes any considerable amount of energy/die space.

  Reply
  - **fgiesen permalink**
    If this was the case, they can't have used it for long; not sure about GF1/2, but GF3 had a FIFO scheme like what I described, as did the Radeons at the time.

    GF3 definitely had slow *static* index buffers (dynamic was fine). IIRC that was because they didn't support index buffers in hardware at all – index data had to come from the command buffer, so the driver needed to `memcpy` index data around.

2. **Corbin Simpson permalink**
   A great article.

   For future reference, you can point people at The X.Org Foundation's repository of documentation, which is all publicly available at http://www.x.org/docs/. For example, AMD's r600/r700 shader documentation, at the assembly level, is publicly viewable at http://www.x.org/docs/AMD/r600isa.pdf. Take it easy!

   Reply
3. **Ignacio permalink**
   I can confirm that your guesses are correct, at least for NVIDIA hardware. Mark Kilgard wrote a bit about the block-based vertex cache in modern GPUs here:

   http://www.slideshare.net/Mark_Kilgard/using-vertex-bufferobjectswell

   It'd be interesting to do some numbers and compare the efficiency of FIFO vs. block caches. For the same size you would find that FIFOs always do a lot better. This is even more so when using primitives with more than 3 vertices. For example, the efficiency when rendering 16×16 patches is terrible, and since vertices are shared by many more patches than triangles…

   Reply
4. **TomF permalink**
   Just to note that every architecture now turns DXasm code into scalar code. "add r0.xyz, r1, r2" turns into three separate instructions, and r0.x, r0.y and r0.z are three completely separate registers. Even the older AMD/ATI architectures that still had VLIW5 or VLIW4 started by scalarising the shader, doing clever things with it, and then re-VLIW'd in cunning ways completely unrelated to how DXasm arranged things. (see the very excellent talk by Norm Rubin "Issues and challenges in compiling for graphics processors", http://dl.acm.org/citation.cfm?id=1356088)

   What does this mean in practical terms? Well first – use the correct write masks – don't rely on the dead-code-elimination of the compiler. Second – nothing is free any more – if you don't need a value, don't calculate it, even if it's "just" the fourth channel. Finally, if you're counting instructions, you always need to multiply by the number of destination targets. "mul r1, r2, r3" is four times as expensive as "mul r1.x, r2, r3". (obvious exceptions for things like dp4 and so on)

Reply

5. **Bala permalink**
   Hi,
   You postings are really interesting. I am using Amd graphics processors have rv730 architecture. 4690. Do you have a good reference docs or material which explains the interenals of this architecture.

   Reply

Trackbacks & Pingbacks

1. (Updated) 3D Graphics Pipeline Explained - 3D Tech News, Pixel Hacking, Data Visualization and 3D Programming - Geeks3D.com
2. A trip through the Graphics Pipeline 2011: Index « The ryg blog
3. A trip through the Graphics Pipeline 2011, part 10 « The ryg blog
4. A trip through the Graphics Pipeline 2011, part 12 « The ryg blog
5. A trip through the Graphics Pipeline 2011, part 13 « The ryg blog