i915 Hardware Contexts (and some bits about batchbuffers)

Author : Ben Widawsky

Disclaimer

First and foremost: Everything documented below is included in the Intel public documentation. Anything I say which offends you are my own words and not those of Intel. (Anything I say which is of monetary value is the opposite).

Daniel asked me to write about hardware contexts on Intel GPUs some time ago. Hardware contexts aren't terribly interesting. However, the role of contexts has been steadily expanding, and as such, you need to start somewhere in the explanation. So let's consider this a precursor to really interesting stuff in the future (I hope).

I use the term, "GPU client" quite a bit. If it helps you, you can think of a specific GPU client:

- mesa
- libva
- X device dependant 2d driver (ddx)

I worked pretty hard to make all SVG images so you can zoom in on your own. Please send me feedback if this didn't work out for you. I know that the arrows turned out stupid, but it works in inkscape, I swear!

Finally, I've yet read Daniel's GEM docs, so maybe some of this is a repeat.

Batchbuffers

I want to take a brief diversion into GPU work. The Direct Rendering Interface enables GPU

clients, ie. userspace libraries such as mesa, to directly submit work to the GPU. Since this flies in the face of every security architect's idea of how to design a system, there was one indirection introduced called a "Batchbuffer." The batchbuffer is nothing more than a set of instructions to be read by the GPU for setting up state, and to a much smaller extent, instructions telling the GPU how to act upon that state. It's special because we can apply some security flags to it, and it's not directly executed by HW. It's indirectly executed via the batchbuffer. The i915 driver has to tell the hardware to execute the batchbuffer, but the i915 driver never modifies the contents of the batchbuffer. Make no mistake, all GPU commands are assembled into this batchbuffer by userspace and they are executed by the GPU. The kernel driver mostly acts as a proxy. Understanding the basic principal of what a batchbuffer does is important in order to understand why hardware contexts are interesting. Batchbuffers, batches, command buffers, and execbuffers, are used interchangeably, and are all the same thing (at least on Intel GPUs, can't speak for others).

Because the system may have many GPU clients, all with the ability to directly render on the HW, it was therefore implicit [historically] that any client which had work to be done by the GPU could not assume any of it's previous GPU state would be preserved upon it's next run. To put it another way, if a GPU client submitted two batchbuffers back to back, it couldn't, and still can't assume that those two run in sequence. For older fixed function hardware, this really didn't matter too much, but that's changing…

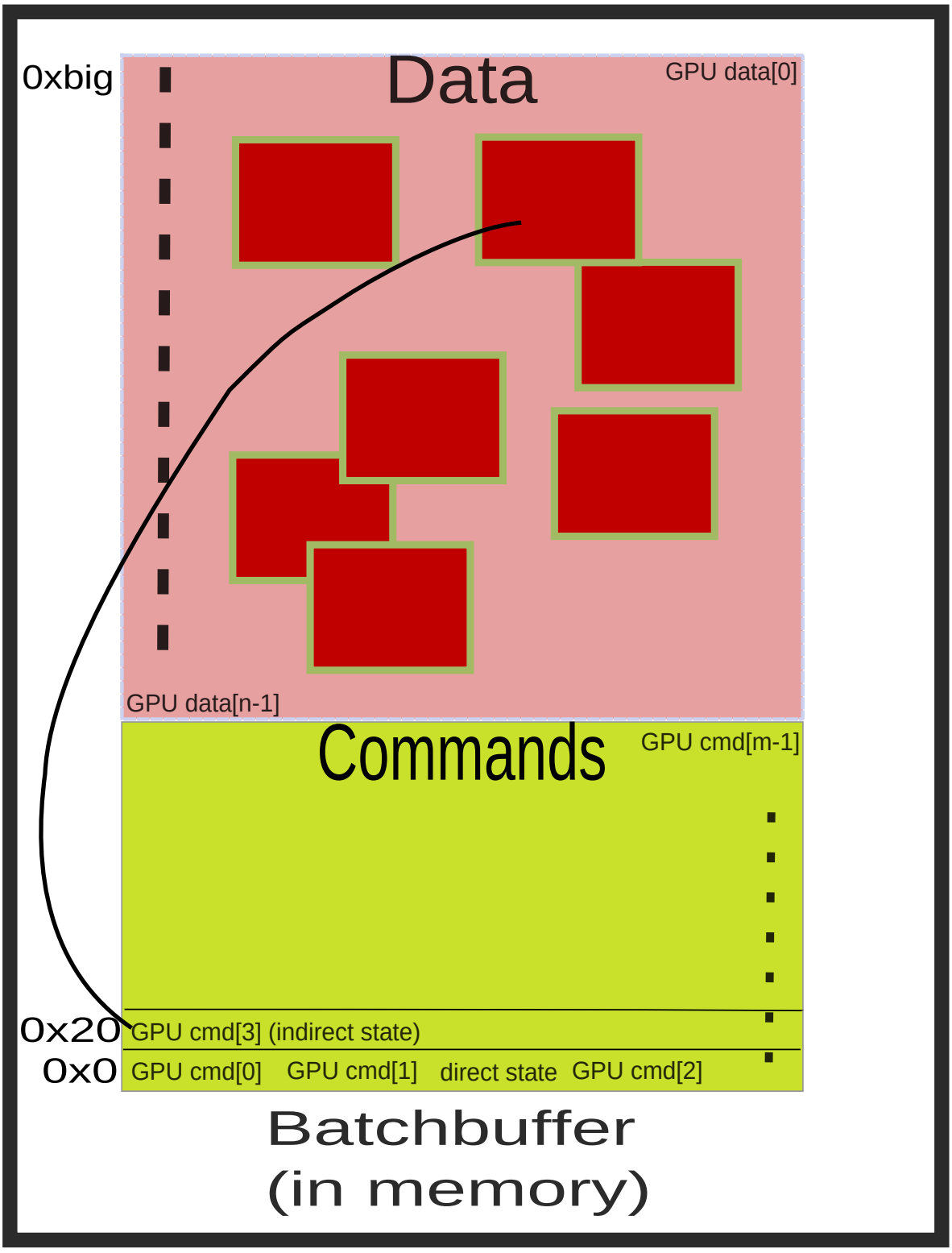
To elaborate a bit more about batchbuffers, a gross oversimplification of how it works: there are 3 (well, 2.5) types of commands,

1. Here is some state, inline with the command (direct state)

2. I have some state, it's located over there (indirect state)

3. Go do something with all that state I gave you (3D_ commands)

Many modern GPU clients build the buffers like a stack/heap. Indirect state goes on the heap, and direct state + commands go on the stack. The commands that have indirect state then point within it’s own buffer. This isn’t strictly necessary, but it makes the allocation easier.


0xbig
Data
GPU data[0]
GPU data[n-1]
Commands
GPU cmd[m-1]
0x20
GPU cmd[3] (indirect state)
0x0
GPU cmd[0] GPU cmd[1] direct state GPU cmd[2]
Batchbuffer
(in memory)

Hardware Contexts

Hardware contexts are a feature which have existed on Intel GEN graphics hardware since Ironlake (GEN5). The Linux kernel's i915 driver has supported it since 3.6 courtesy of yours truly :-D. Much the same as CPUs, it allows GPUs to save off certain state and restore that state at a later time. This is interesting primarily for two reasons:

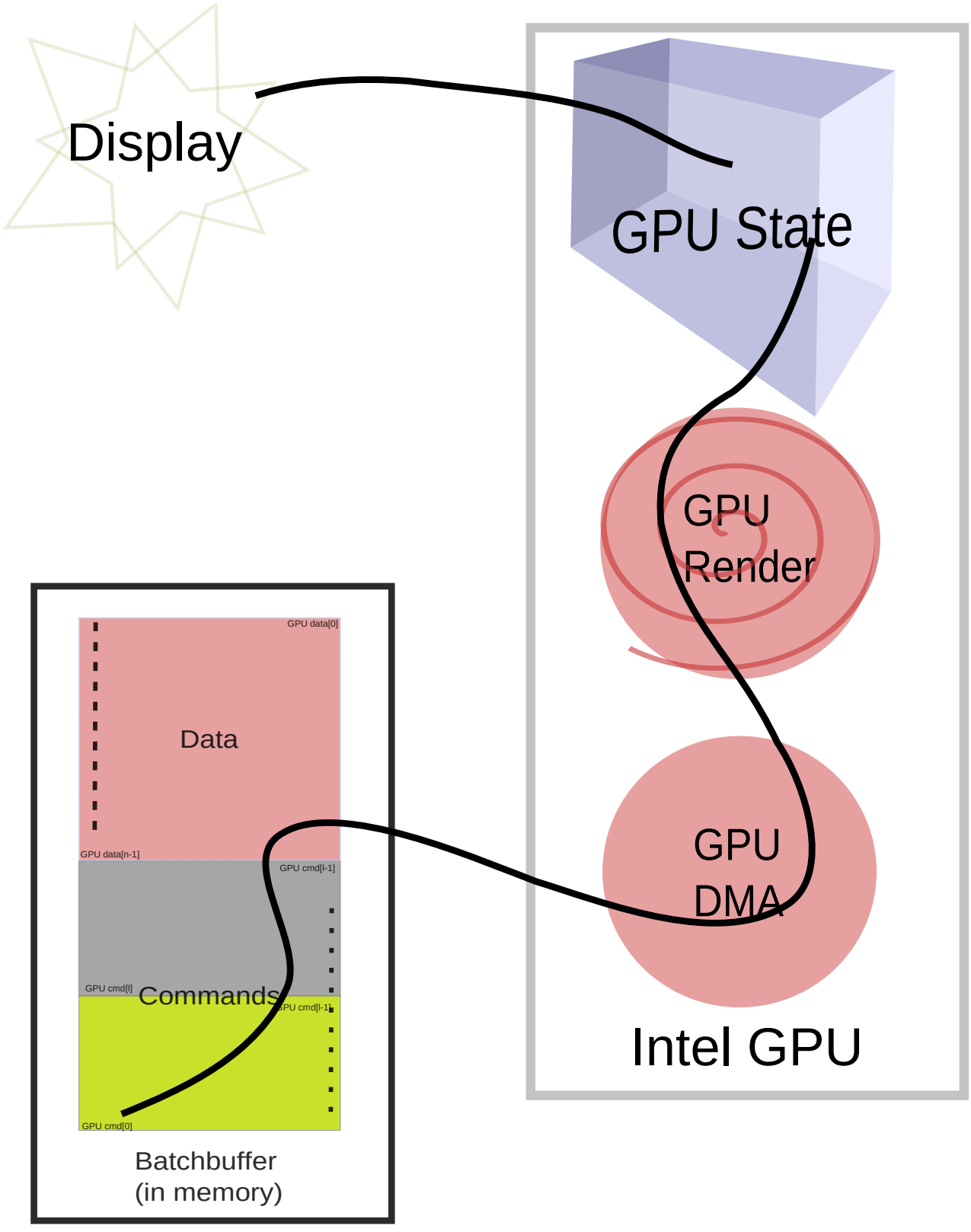
1. It potentially reduces the overhead on clients having to restore their context.

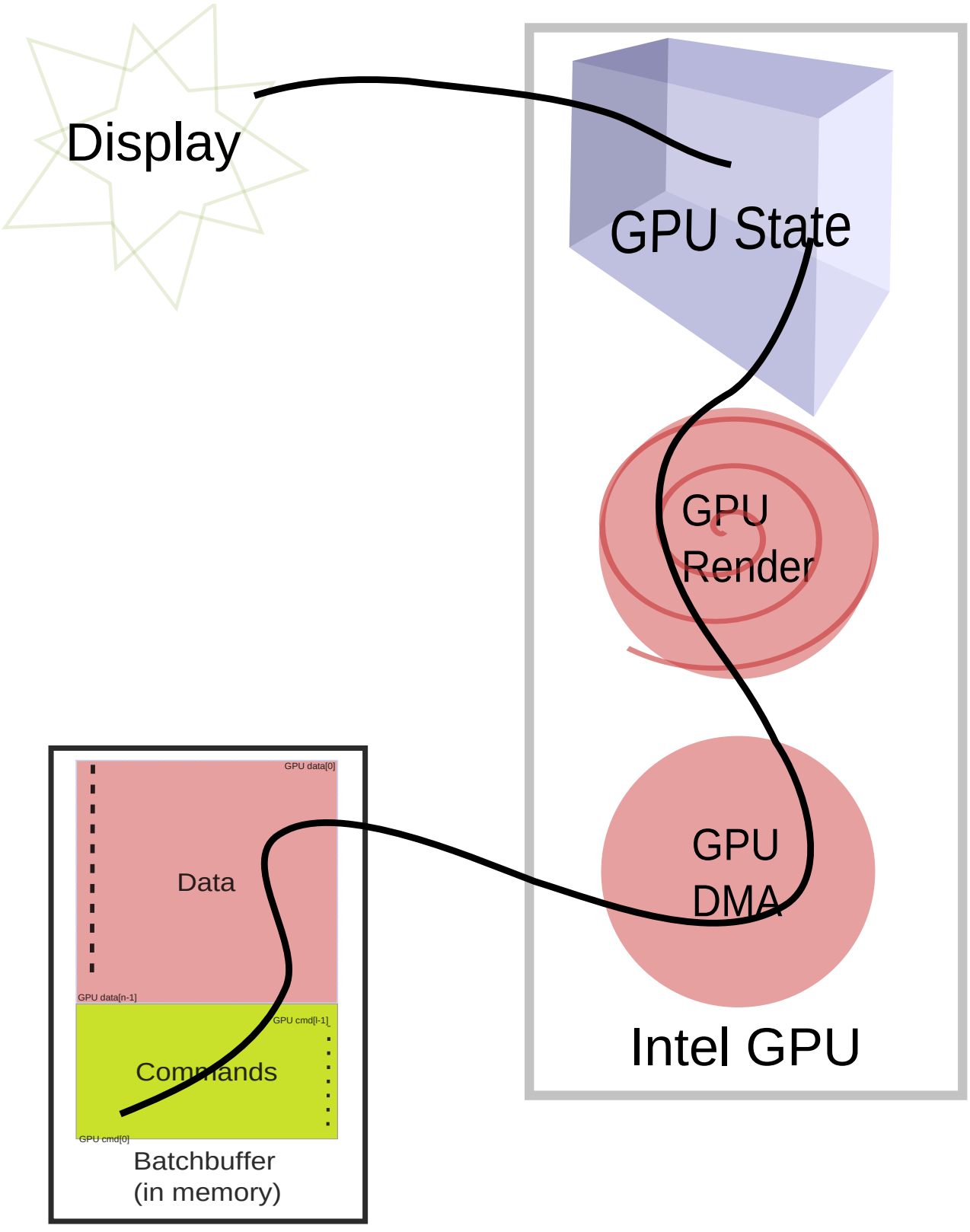
2. It allows GPU clients to save their context ie. recover GPU state from a previous batch. I'll explain both in more detail.

Reducing overhead

Reducing overhead is a fairly straightforward thing, and like many simple optimizations, this one doesn't buy us much. GPU batchbuffers (AKA execbufs, or command buffers) do not need to emit as many commands to the GPU since some of it's state was saved from a previous set of commands. This both reduces the amount of DMA traffic, as well as the number of commands the GPU has to execute (the latter is arguable because it's HW implementation specific).

Some pictures! The first picture shows without contexts, and the second, with contexts. The only thing to observe is that with contexts, the batch buffer size is slightly smaller.


Display
GPU State
GPU
Render
GPU
DMA
Intel GPU
GPU data[0]
Data
GPU data[n-1]
GPU cmd[i-1]
GPU cmd[i]
Commands
GPU cmd[i-1]
GPU cmd[0]
Batchbuffer
(in memory)

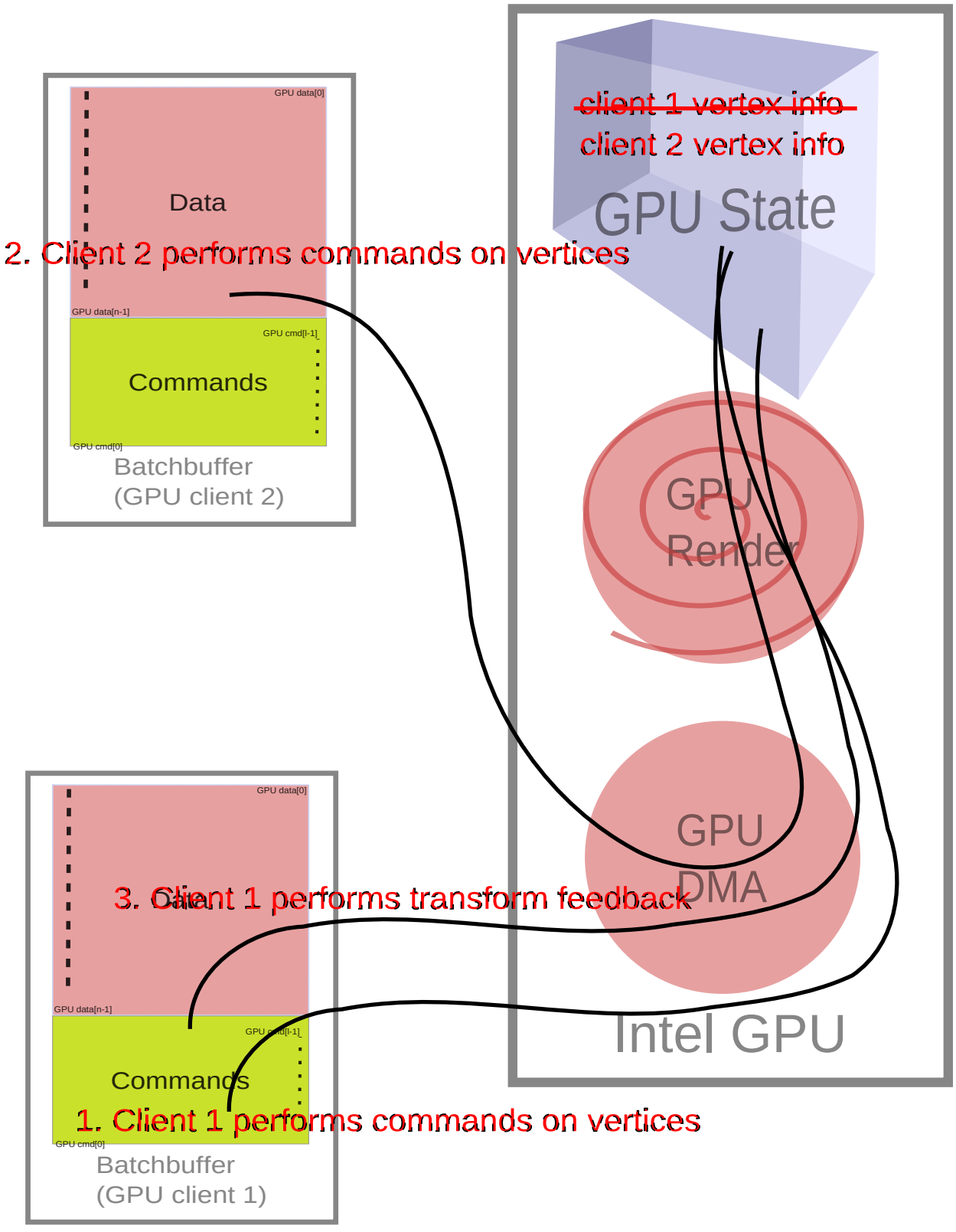

Display
GPU State
GPU
Render
GPU
DMA
Intel GPU
GPU data[0]
Data
GPU data[n-1]
GPU cmd[l-1]
Commands
GPU cmd[0]
Batchbuffer
(in memory)

Since much of the state setup must happen on every batch for reasons which are mostly out of scope, we don’t get a huge savings here – though nobody has actually measured it.

Preserving State

The second reason for contexts is the ability for a GPU client to have its context restored. This featured is actually required in order to implement modern OpenGL.

Recall earlier where I stated that a GPU client can never depend on two batches executing consecutively. That matters if the second operation relied on results from GPU operations in the first batch. If mesa submits some vertices to be processed, and then the DDX for example goes and wipes out all the GPU state, how can mesa then depend on any GPU state. It’s already been flushed through the GPU pipeline by whatever the DDX was doing.


GPU data[0]
Data
GPU data[n-1]
GPU cmd[l-1]
Commands
GPU cmd[0]
Batchbuffer
(GPU client 2)
client 1 vertex info
client 2 vertex info
GPU State
2. Client 2 performs commands on vertices
GPU
Render
GPU
DMA
3. Client 1 performs transform feedback
Intel GPU
GPU data[0]
Data
GPU data[n-1]
GPU cmd[l-1]
Commands
GPU cmd[0]
1. Client 1 performs commands on vertices
Batchbuffer
(GPU client 1)

As demonstrated above, badly, client 1 requests some work to be done from the GPU and the intermediate pipeline output is stored somewhere in the GPUs internal state. Well, once client 2 executes, it uses the same pipeline, and so that info is lost. So in step 3. when client 1 again submits a batch to query some info from the pipeline state, it would read back client 2?s state.

Transform feedback was the first immediate need for HW contexts' ability to preserve state, and is still currently the best example of this. GEN hardware implement the feedback buffer via a pointer (to the buffer) and an offset denoting the next free space within the buffer (within the buffer). The offset can be reset, but not arbitrarily moved by software. As information is dumped into the feedback buffer, the offset is incremented by the hardware. That offset is saved within the HW context state. Keeping the offset around is important because you don't want to destroy the previous feedback data. To date, this is actually achievable simply by the fact that we do not implement Geometry Shaders. The number of vertices, and therefore the amount of vertex information (and thus, the offset) can be accurately predicted and accounted for. On the next batch you set the pointer to where the calculated offset would have been, reset the offset, and away you go.

Geometry Shaders create an unpredictable number of new vertices, and because of that, the above hack will no longer work.

Kernel Implementation

The kernel implementation isn't very interesting IMO, and I can write a separate article if anyone requests one. There are a couple of gotchas in the implementation, like the initial state, as an example, but the concept is very straight forward. As I mentioned earlier in regards to Direct Rendering, GPU clients do not submit their work directly. The clients submit work to the kernel, which then tells the GPU to consume that work. This gives the kernel the opportunity to insert the proper save and restore commands to the GPU without the clients knowing. For example, if we have two clients X, and Y it might be something like:

```
  ... <gotchas>  client X submits batch  i915 emits X's batch to hardw
are  client Y submits batch  i915 (via HW context) saves client X's st
ate, and restore Y's state  i915 emits Y's batch to hardware
```