The Anatomy of a Vulkan™ Driver

Jason Ekstrand

intel Software
Intel OpenSource TECHNOLOGY CENTER

Obligatory Brag Side

Between May 8, 2015 and early Febuary 2016, we delivered:

- A brand new, from scratch*, driver
- Against a brand-new API
- With only 3-3.5 (average) people
- In 8 months
- We were conformant on Day 1
- We were open-source on Day 1

* When initially released, only the back-end compiler and core NIR were shared.

Huge thanks to:

Kristian Høgsberg Kristensen
Chad Versace
Nanley Chery
Jordan Justen
Mark Janes
Dylan Baker
Connor Abbott
Kenneth Graunke
Anuj Phogat
Lionel Landwerlin
The Igalia team
Francisco (Curro) Jerez
Ben Widawsky

What is the Vulkan API?

Vulkan is a new 3-D rendering and compute api from Khronos, the same cross-industry group that maintains OpenGL

- Redesigned from the ground-up; It is *not* OpenGL++
- Designed for modern GPUs and software
- Will run on currently shipping (GL ES 3.1 class) hardware

Why do we need a new 3-D API?

- OpenGL 1.0 was released by SGI in January of 1992
  - Based on the proprietary IRIS GL API
- Brian Paul released mesa in August of 1993
- Computers have advanced a lot in 24 years:
  - GPUs are more powerful and flexible
  - Memory has gotten cheaper
  - Multi-core CPUs are common
- OpenGL has done amazingly well over the last 24 years!

Why do we need a new 3-D API?

Not everything in OpenGL has stood the test of time:

- The OpenGL is API is a state machine
- OpenGL state is tied to a single on-screen context
- OpenGL hides *everything* the GPU is doing

This all made sense in 1992!

Why do we need a new 3-D API?

Much has changed since 1992:

- Multithreading is now common-place
  - A state machine based on a singleton context doesn't thread well
- Off-screen rendering is a thing
  - Why do I need to talk to X11 to get a context?
- GPU hardware is much more standardized
  - You don't ***need*** to hide everything
  - App developers ***don't want*** you to hide everything

OpenGL has adapted as well as it can

Why do we need a new 3-D API?

Vulkan takes a different approach:

- Vulkan is an object-based API with no global state
  - All state concepts are localized to a command buffer
- WSI is an extension of Vulkan, not the other way round.
- Vulkan far more explicit about what the GPU is doing
  - Texture formats, memory management, and syncing are client-controlled
  - Enough is hidden to maintain cross-platform compatibility
- Vulkan drivers do no error checking!

The Anatomy of a Vulkan Driver

Let's look at `vkCmdDraw`...

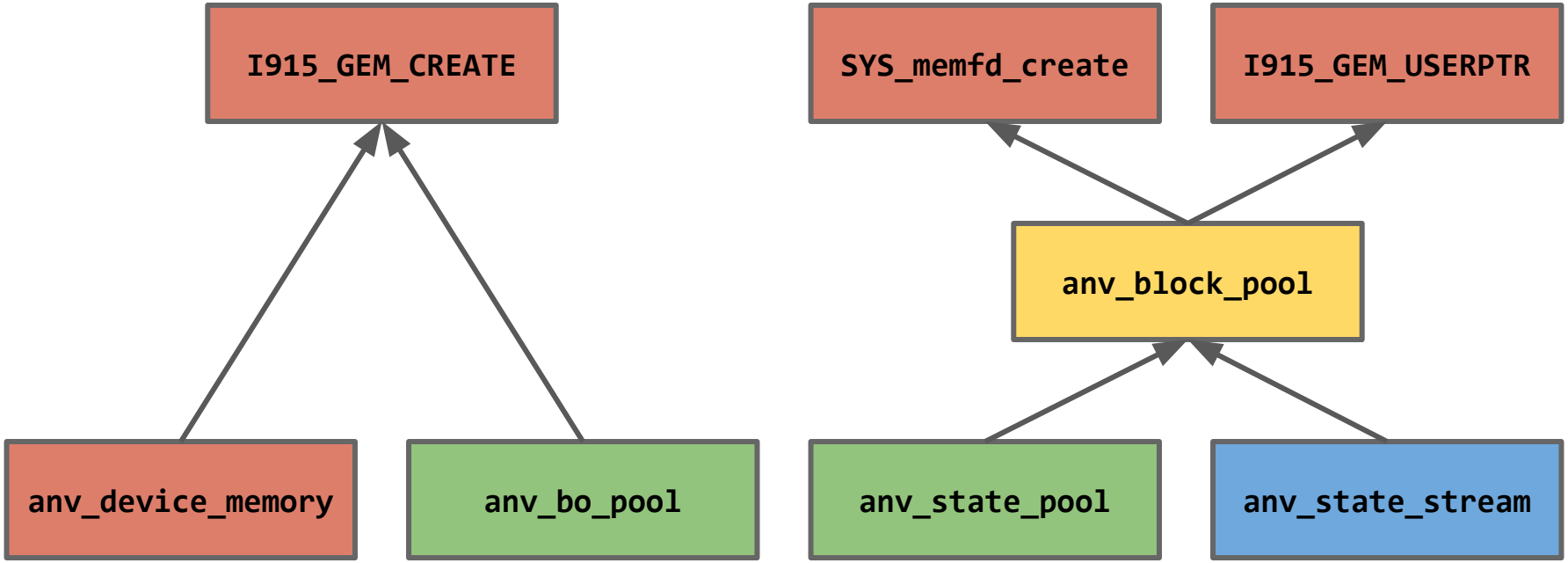
GPU Memory Allocation

Vulkan provides much more explicit control of memory allocation

- Users are presented with a collection of “heaps”
- From those heaps, they allocate VkDeviceMemory objects
- VkImage and VkBuffer objects are placed at explicit offsets within a VkDeviceMemory object (client-controlled sub-allocation).
- Other objects have small bits of driver-allocated memory:
  - VkImageView, VkCmdBuffer, VkQueryPool, etc.

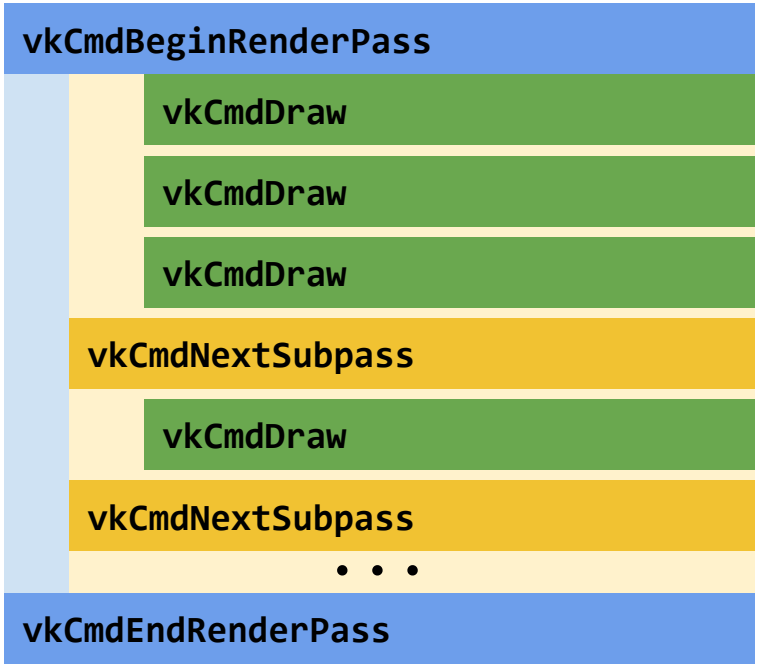
VkDeviceMemory maps nicely to a GEM bo but not to libdrm

GPU Memory Allocation



Compression & Resolves

- Most hardware has some form of on-the-fly compression:
  - Compressed MSAA
  - HiZ for Depth
  - CCS for single-sampled color
- Require “resolves”
- Can’t do CPU-side tracking
- Vulkan provides render passes and layout transitions



Is it easier to write a Vulkan driver?

Yes, very much so...

- No error checking!
- No vkVertex4f or polygon stipples
- SPIR-V is a *little* easier to handle than GLSL
- The Vulkan CTS is ~115k tests you don't have to write

But some things are harder:

- No CPU-side object state tracking
- Apps have more power for stupid

Code sharing between Vulkan and GL

There are a few different options:

- Mega-API approach (i.e. gallium):
  - Vulkan and GL on Gallium
  - GL on Vulkan
  - GL and Vulkan on a new api (let’s call it Helium)
- Duplicate impunity
- Toolbox approach:
  - A bunch of different pieces that can be assembled into a driver
  - Similar to the way that NIR is designed

Code sharing between Vulkan and GL

The Intel driver-building toolbox:

`src/intel/common/`	Misc. common code
`src/intel/genxml/`	Autogenerated state packet fill-out code
`src/intel/isl/`	Surface layout calculations
`src/intel/blorp/`	Blit, clear, and resolve framework
`src/compiler/nir/`	Core compiler infrastructure
`src/intel/compiler/*`	Back-end shader compiler

* The compiler has yet to be moved. It still lives in src/mesa/drivers/dri/i965/ at the moment

Questions?