Achieving High-performance Graphics on Mobile With the Vulkan API

ARM

Marius Bjørge
Graphics Research Engineer

GDC 2016

Agenda

- Overview
- Command Buffers
- Synchronization
- Memory
- Shaders and Pipelines
- Descriptor sets
- Render passes
- Misc

Overview – OpenGL

- OpenGL is mainly single-threaded
  - Drawcalls are normally only submitted on main thread
  - Multiple threads with shared GL contexts mainly used for texture streaming

- OpenGL has a lot of implicit behaviour
  - Dependency tracking of resources
  - Compiling shader combinations based on render state
  - Splitting up workloads
  - All this adds API overhead!

- OpenGL has quite a small footprint in terms of lines of code

Overview – Vulkan

- Vulkan is designed from the ground up to allow efficient multi-threading behaviour

- Vulkan is explicit in nature
  - Applications must track resource dependencies to avoid deleting anything that might still be used by the GPU or CPU
  - Little API overhead

- Vulkan is very verbose in terms of lines of code
  - Getting a simple "Hello Triangle" running requires ~1000 lines of code

Overview

- To get the most out of Vulkan you probably have to think about re-designing your graphics engine

- Migrating from OpenGL to Vulkan is not trivial

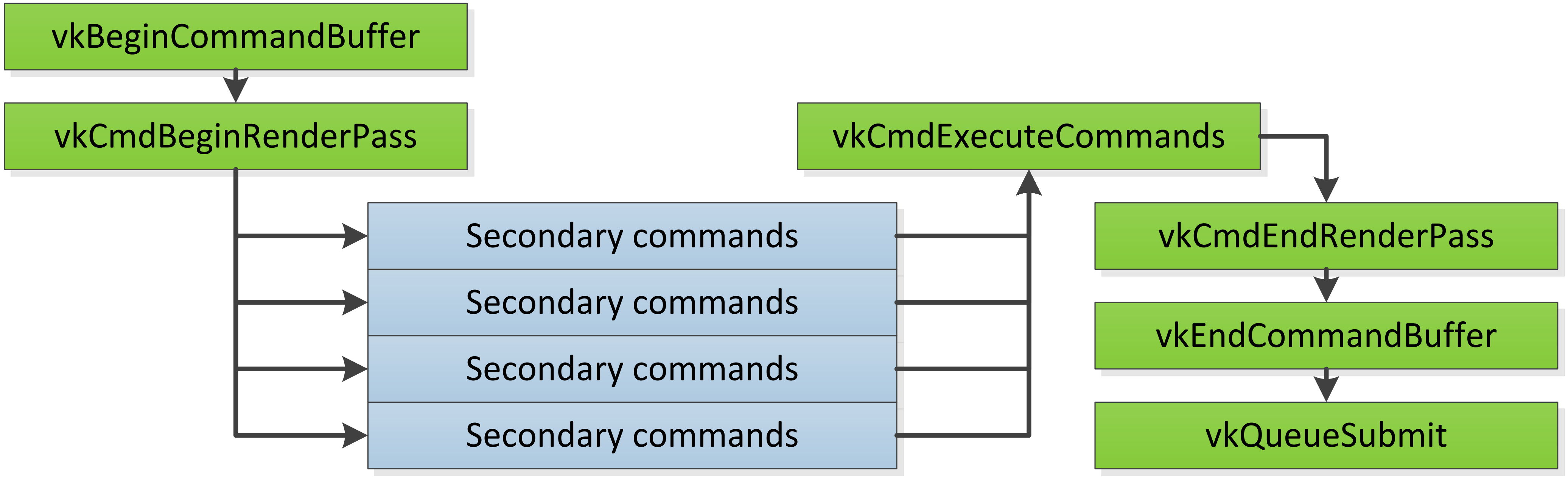
- Some things to keep in mind:
  - What performance level are you targeting?
  - Do you really need Vulkan?
  - How important is OpenGL support?
  - Portability?

Command Buffers

- Used to record commands which are later submitted to a device for execution
  - This includes draw/dispatch, texture uploads, etc.
- Primary and secondary command buffers

- Command buffers work independently from each other
  - Contains all state
  - No inheritance of state between command buffers

Command Buffers



Command Buffers

- In order to have a common higher-level command buffer abstraction we also had to support the same interface in OpenGL
  - Record commands to linear allocator and playback later
  - Uniform data pushed to a separate linear allocator per command buffer

Synchronization

- Submitted work is completed out of order by the GPU
- Dependencies must be tracked by the application
  - Using output from a previous render pass
  - Using output from a compute shader
  - Etc
- Synchronization primitives in Vulkan
  - Pipeline barriers and events
  - Fences
  - Semaphores

Allocating Memory

- Memory is first allocated and then bound to Vulkan objects
  - Different Vulkan objects may have different memory requirements
  - Allows for aliasing memory across different vulkan objects
- Driver does no ref counting of any objects in Vulkan
  - Cannot free memory until you are sure it is never going to be used again

- Most of the memory allocated during run-time is transient
  - Allocate, write and use in the same frame
  - Block based memory allocator

Block Based Memory Allocator

- Relaxes memory reference counting
- Only entire blocks are freed/recycled

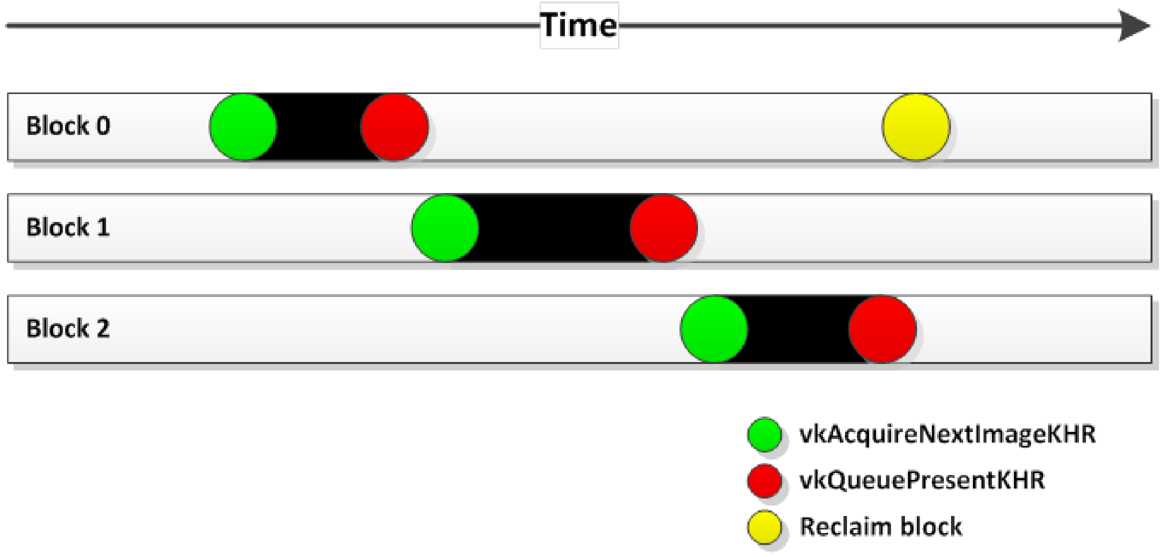


Image Layout Transitions

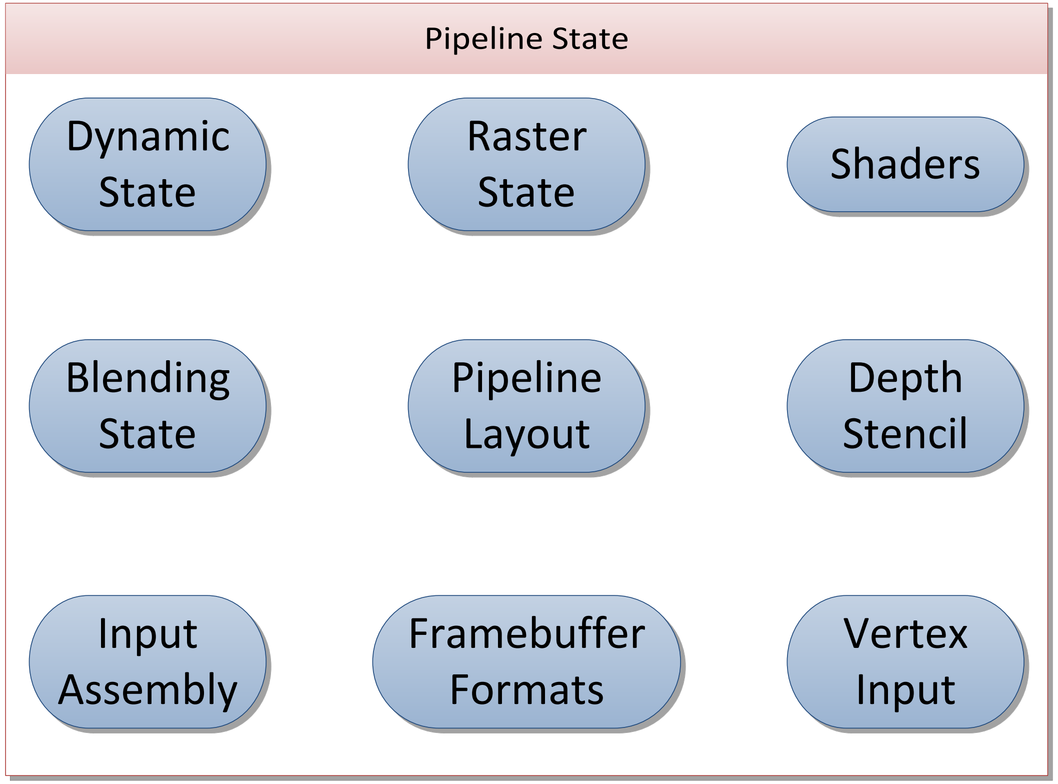
- Must match how the image is used at any time
- Pedantic or relaxed
  - Some implementations might require careful tracking of previous and new layout to achieve optimal performance
  - For Mali we can be quite relaxed with this – most of the time we can keep the image layout as VK_IMAGE_LAYOUT_GENERAL

Pipelines

- Vulkan bundles state into big monolithic pipeline state objects
- Driver has full knowledge during shader compilation

```
vkCreateGraphicsPipelines(...)
;

vkBeginRenderPass(...);
vkCmdBindPipeline(pipeline);
vkCmdDraw(...);
vkEndRenderPass(...);
```

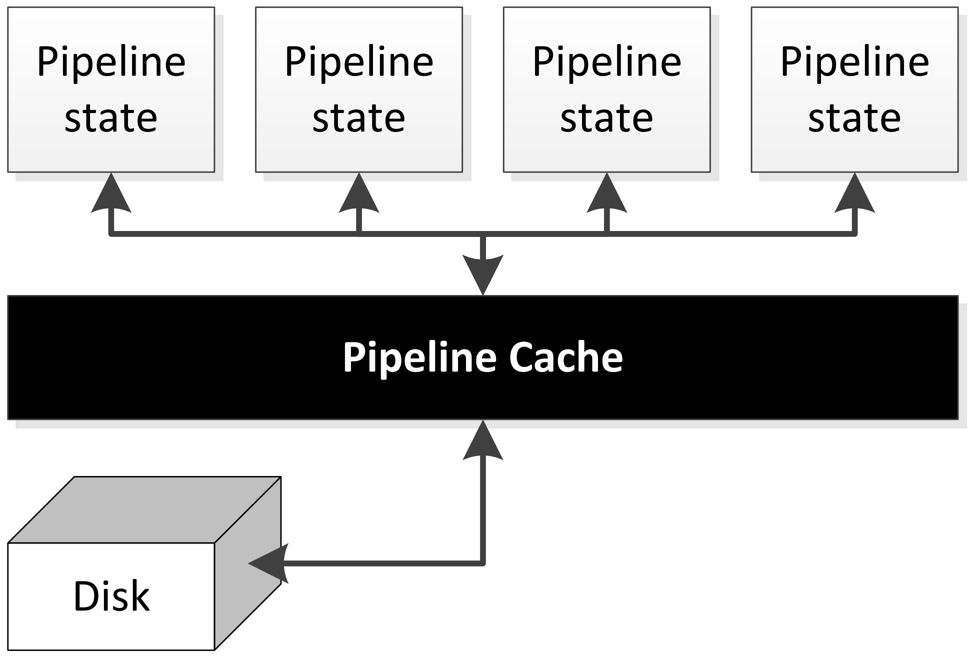


Pipelines

- In an ideal world…
  - All pipeline combinations should be created upfront
- …but this requires detailed knowledge of every potential shader/state combination that you might have in your scene
  - As an example, one of our fragment shaders has ~9 000 combinations
  - Every one of these shaders can use different render state
  - We also have to make sure the pipelines are bound to compatible render passes
  - An explosion of combinations!

Pipeline Cache

- Result of the pipeline construction can be re-used between pipelines
- Can be stored out to disk and re-used next time you run the application



Shaders

- Vulkan standardized on SPIR-V
- No more pain with GLSL compilers behaving differently between vendors?

- Khronos reference compiler
  - GL_KHR_vulkan_glsl
  - Library that can be integrated into your graphics engine
  - Can output SPIR-V from GLSL

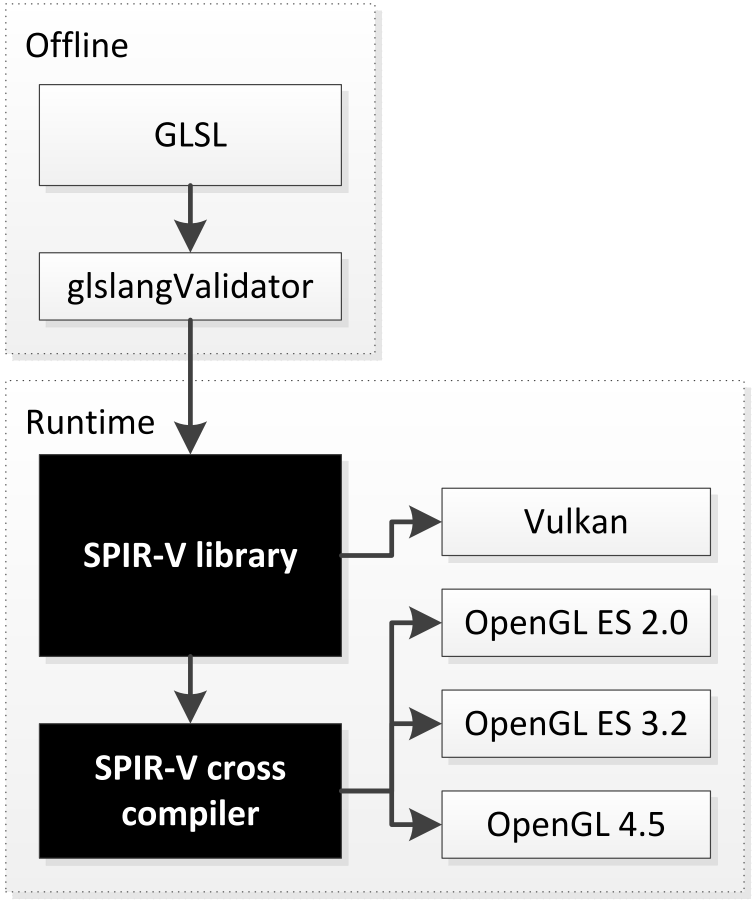
- We decided early to internally standardize the engine on SPIR-V
  - Use SPIR-V cross compiler to output GLSL

SPIR-V

- Why SPIR-V?
  - The SPIR-V ecosystem is currently very small – but we anticipate that this will change over the coming years as we are already seeing optimization tools in progress on github.

- SPIR-V cross compiler
  - We wrote this library in order to parse and cross compile SPIR-V binary source
  - Is available as open source on <INSERT LOCATION>
  - (…or hoping to open-source this at some point)

Shaders



Offline
GLSL
glslangValidator
Runtime
SPIR-V library
SPIR-V cross compiler
Vulkan
OpenGL ES 2.0
OpenGL ES 3.2
OpenGL 4.5

SPIR-V

- Using SPIR-V directly we can retrieve information about bindings as well as inputs and outputs
  - This is useful information when creating or re-using existing pipeline layouts and descriptor set layouts
  - Also allows us to easily re-use compatible pipeline layouts across a bunch of different shader combinations
    - Which also means fewer descriptor set layouts to maintain

Descriptor Sets

- Textures, uniform buffers, etc. are bound to shaders in descriptor sets
  - Hierarchical invalidation
  - Order descriptor sets by update frequency

- Ideally all descriptors are pre-baked during level load
  - Keep track of low level descriptor sets per material…
  - …but, this is not trivial

- Our solution:
  - Keep track of bindings and update descriptor sets when necessary

Descriptor Sets

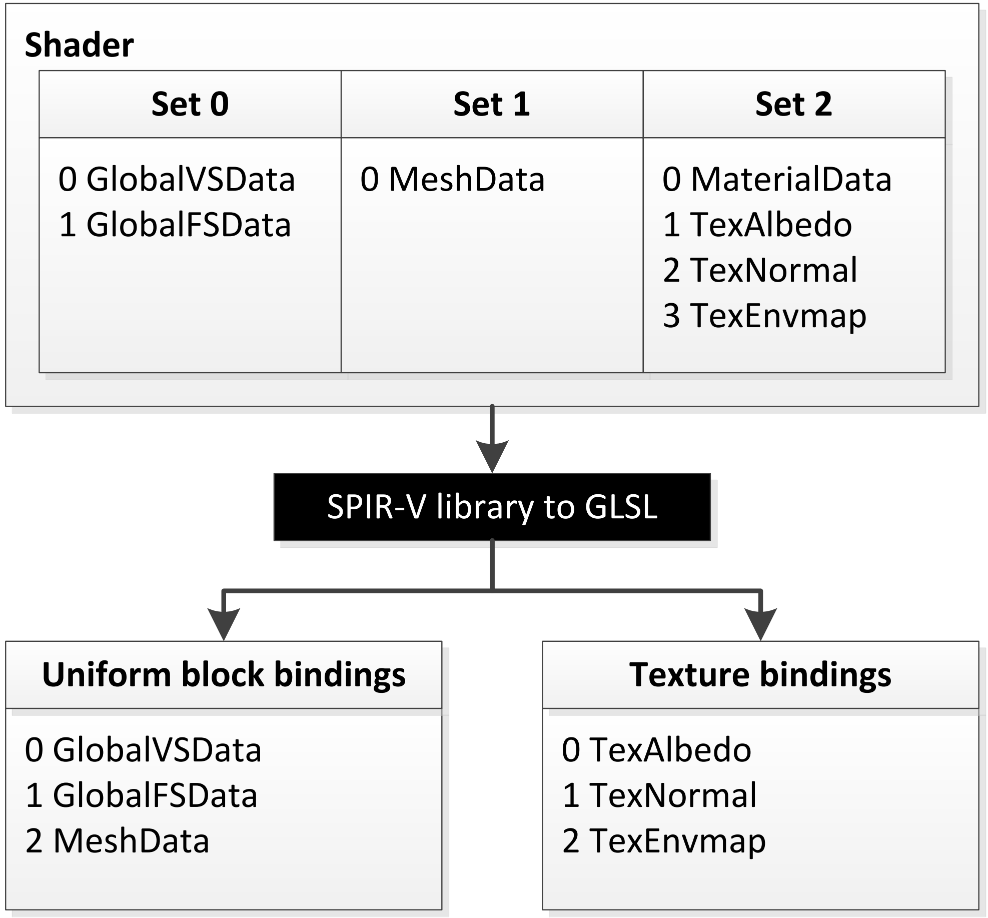
```
layout (set=0, binding=0) uniform ubo0
{
  // data
};
layout (set=0, binding=1) uniform sampler2D TexA;
layout (set=1, binding=0) uniform sampler2D TexB;
layout (set=1, binding=2) uniform sampler2D TexC;
```


Descriptor Set Emulation

- We also need to support this in OpenGL

- Our solution:
  - Added support for emulating descriptor sets in our OpenGL backend
  - Use SPIR-V cross compiler library to collapse and serialize bindings

Descriptor Set Emulation



Push Constants

- Push constants replace non-opaque uniforms
  - Think of them as small, fast-access uniform buffer memory
- Update in Vulkan with vkCmdPushConstants
- Directly mapped to registers on Mali GPUs

```
// New
layout(push_constant, std430) uniform PushConstants {
    mat4 MVP;
    vec4 MaterialData;
} RegisterMapped;

// Old, no longer supported in Vulkan GLSL
uniform mat4 MVP;
uniform vec4 MaterialData;
```

Push Constant Emulation

- Again, we need to support OpenGL as well

- Our solution:
  - Use SPIR-V cross compiler to turn push constants into regular non-opaque uniforms
  - Logic in our OpenGL/Vulkan backends redirect the push constant data appropriately

Render Passes

- Knowing when to keep and when to discard
- Render passes in Vulkan are very explicit
  - Declare when a render pass begins
    - Load, discard or clear the framebuffer?
  - Declare when a render pass ends
    - Which parts do you need to be committed to memory?

Subpass Inputs

- Vulkan supports subpasses within render passes
- Standardized GL_EXT_shader_pixel_local_storage!

```
// GLSL
#extension GL_EXT_shader_pixel_local_storage : require
__pixel_local_inEXT GBuffer {
    layout(rgba8) vec4 albedo;
    layout(rgba8) vec4 normal;
    ...
} pls;

// Vulkan
layout(input_attachment_index = 0) uniform subpassInput albedo;
layout(input_attachment_index = 1) uniform subpassInput normal;
...
```

Subpass Input Emulation

- Supporting subpasses in GL is not trivial, and probably not feasible on a lot of implementations

- Our solution:
  - Use the SPIR-V cross compiler library to rewrite subpass inputs to Pixel Local Storage variables
  - This will only support a subset of the Vulkan subpass features, but good enough for our current use

Misc

- Yet another coordinate system
  - Similar to D3D except Y direction in clip-space is inverted
    - Simple solution: Invert gl_Position.y in your vertex shaders
    - …or use swapchain transform if the driver supports it

- Mipmap generation
  - No equivalent glGenerateMipmaps() in Vulkan
  - Roll your own using vkCmdBlitImage()

Thank you!

ARM

To Find Out More….

ARM Booth #1624 on Expo Floor:

- Live demos of the techniques shown in this session
- In-depth Q&A with ARM engineers
- More tech talks at the ARM Lecture Theatre

http://malideveloper.arm.com/gdc2016:

- Revisit this talk in PDF and video format post GDC
- Download the tools and resources

More Talks From ARM at GDC 2016

Available post-show at the Mali Developer Center: malideveloper.arm.com/

Vulkan on Mobile with Unreal Engine 4 Case Study
Weds. 9:30am, West Hall 3022

Making Light Work of Dynamic Large Worlds
Weds. 2pm, West Hall 2000

Achieving High Quality Mobile VR Games
Thurs. 10am, West Hall 3022

Optimize Your Mobile Games With Practical Case Studies
Thurs. 11:30am, West Hall 2404

An End-to-End Approach to Physically Based Rendering
Fri. 10am, West Hall 2020