Porting to Vulkan

Hans-Kristian Arntzen
Engineer, ARM
(Credit for slides: Marius Bjørge)
Agenda

• API flashback
• Engine design
  - Command buffers
  - Pipelines
  - Render passes
  - Memory management
API Flashback

![Diagram comparing OpenGL and Vulkan with a logic shift](image.png)
API Flashback
Porting from OpenGL to Vulkan?

- Most graphics engines today are designed around the principles of implicit driver behaviour
  - A direct port to Vulkan won’t necessarily give you a lot of benefits

- Approach it differently
  - Re-design for Vulkan, and then port that to OpenGL
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
  - Also applies to API handles!

- 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
- Sub-allocations take refcount on block
Command Buffers

- Request command buffers on the fly
  - Allocated using ONE_TIME_SUBMIT_BIT
  - Recycled

- Separate command pools per
  - Thread
  - Frame
  - Primary/secondary
Secondary Command Buffers

Main thread

Thread 0

Thread 1

Thread 2

vkCommandPool → vkCommandBuffer

vkBeginRenderPass

vkCmdExecuteCommands

vkEndRenderPass

Secondary command buffer
Shaders

- Standardize on SPIR-V binary shaders
- Extensively use the Khronos SPIRV-Cross library
  - Cross compiling back to GLSL
  - Provides shader reflection for
    - Vertex attributes
    - Subpass attachments
    - Pipeline layouts
    - Push constants
Pipelines

<table>
<thead>
<tr>
<th>Pipeline state</th>
<th>Dynamic state</th>
<th>Shaders</th>
<th>Render pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blend State</td>
<td></td>
<td>Pipeline layout</td>
<td></td>
</tr>
<tr>
<td>Rasterizer state</td>
<td></td>
<td>Vertex input</td>
<td></td>
</tr>
<tr>
<td>Depth/stencil state</td>
<td></td>
<td>Input assembly</td>
<td></td>
</tr>
</tbody>
</table>
Pipelines

- Not trivial to create all required pipeline state objects upfront

- Our approach:
  - Keep track of all pipeline state per command buffer
  - Flush pipeline creation when required
    - In our case this is implemented as an async operation
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 have ~9000 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

- Vulkan has built-in support for pipeline caching
  - Store to disk and re-use on next run

- Can also speed up pipeline creation during run-time
  - If the pipeline state is already in the cache it can be re-used
Pipeline layout

- Defines what kind of resources are in each binding slot in your shaders
  - Textures, samplers, buffers, push constants, etc
- Can be shared among different pipeline objects
Pipeline layout

- Use SPIRV-Cross to automatically get binding information from SPIR-V shaders
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
Descriptor Sets

• Our solution:
  - Keep track of bindings and update descriptor sets when necessary
  - Keep cache of descriptor sets used with immutable Vulkan objects
Descriptor Set emulation

• We also need to support this in OpenGL

• Our solution:
  - Emulate descriptor sets in our OpenGL backend
  - SPIRV-Cross collapses and serializes bindings
Descriptor Set emulation

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 GlobalVSData 1 GlobalFSData</td>
<td>0 MeshData</td>
<td>0 MaterialData 1 TexAlbedo 2 TexNormal 3 TexEnvmap</td>
</tr>
</tbody>
</table>

SPIR-V library to GLSL

**Uniform block bindings**
- 0 GlobalVSData
- 1 GlobalFSData
- 2 MeshData

**Texture bindings**
- 0 TexAlbedo
- 1 TexNormal
- 2 TexEnvmap
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

```cpp
// 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

• But again, we need to support OpenGL as well

• Our solution:
  - Use SPIRV-Cross to turn push constants into regular non-opaque uniforms
  - Logic in our OpenGL/Vulkan backends redirect the push constant data appropriately
Render pass

- Used to denote beginning and end of rendering to a framebuffer

- Can be re-used but must be compatible
  - Attachments: Framebuffer format, image layout, MSAA?
  - Subpasses
  - Attachment load/store
Subpass Inputs

- Vulkan supports subpasses within render passes
- Standardized GL_EXT_shader_pixel_local_storage!
- Also useful for desktop GPUs
Subpass Input Emulation

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

• Our solution:
  - Use SPIRV-Cross to rewrite subpass inputs to Pixel Local Storage variables or texture lookups
  - This will only support a subset of the Vulkan subpass features, but good enough for our current use
Synchronization

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

- Most of the time the application knows upfront how the output of a renderpass is going to be used afterwards
- Internally we have a couple of usage flags that we assign to a render pass
  - On EndRenderPass we implicitly trigger a pipeline barrier
Image Layout Transitions

- Must match how the image is used at any time
- Pedantic or relaxed
  - Some implementations will 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
Summary

• Don’t allocate or release during runtime
• Batching still applies
• Multi-thread your code!
• Use push-constants as much as possible
• Multi-pass is fantastic on mobile GPUs