Vulkan Subpasses
or
The Frame Buffer is Lava

Andrew Garrard
Samsung R&D Institute UK
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
  - Efficient generation of work on multiple CPU cores

![Diagram showing command buffer recording]
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
- Efficient generation of work on multiple CPU cores
- Reuse of command buffers to avoid CPU build time
- Potentially more efficient memory management
Vulkan: Making use of the GPU more efficient

• Vulkan aims to reduce the overheads of keeping the GPU busy
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
  - Potentially more efficient memory management
  - Avoiding unpredictable shader compilation
Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
  - Potentially more efficient memory management
  - Avoiding unpredictable shader compilation

- Mostly, the message has been that if you’re entirely limited by shader performance or bandwidth, Vulkan can’t help you (there is no magic wand)
Vulkan: Making use of the GPU more efficient

• Actually, that’s not entirely true...

• APIs like OpenGL were designed when the GPU looked very different (or was partly software)

• The way to design an efficient mobile GPU is not a perfect match for OpenGL
  - Think a CPU’s command decode unit/microcode

• But the translation isn’t always perfectly efficient
Tiled GPUs

• Most (not all) mobile GPUs use tiling
  - It’s all about the bandwidth (size and power limits)

• On-chip tile memory is much faster than the main frame buffer
Not everything reaches memory

• Rendering requires lots of per-pixel data
  - Z, stencil
  - Full multisample resolution

• We usually only care about the final image
  - We can throw away Z and stencil
  - We only need a downsampled (A)RGB
  - Don’t need to load anything from a previous frame
Sometimes we want the results of rendering

- Output from one rendering job can be used by the next
- Z buffer for shadow maps
- Rendering for environment maps
- HDR bloom

- These can have low resolution and may not take much bandwidth
Sometimes you *do* need framebuffer resolution

**Deferred shading**

- Light weight render storing per-surface content at each fragment
- Render full-screen quad and perform fragment shading
- Z
- Diffuse/\(a\)
- Specular/Specularity
- Normal
Sometimes you *do* need framebuffer resolution

- Deferred shading
- Deferred lighting
Sometimes you do need framebuffer resolution

- Deferred shading
- Deferred lighting
- Order-independent transparency
Sometimes you do need framebuffer resolution

- Deferred shading
- Deferred lighting
- Order-independent transparency
- HDR tone mapping
Rendering outputs separately

• Rendering to each surface separately is *bad*

• Geometry has a per-bin cost
  - Sometimes the cost is low, but it’s there
  - Vertices in multiple bins get processed repeatedly
  - Rendering the scene repeatedly is painful

• Even immediate-mode renderers hate this!
Multiple render targets don’t help much

• Using MRTs means multiple buffers in one pass
  - This is a typical approach for immediate-mode renderers (e.g. desktop/console systems)

• Reduces the geometry load (only process once)

• Still writing a *lot* of data off-chip
  - Tilers are all about trying not to do this!
  - Increases use of shader resources may slow some h/w
Pixel Local Storage (OpenGL ES extension)

- **Tiler-friendly (at last)**
  - Store only the current tile values
  - Read them later in the tile processing

- **But not portable!**
  - Not practical on immediate renderers
  - Debugging on desktop won’t work!
  - Capabilities vary between devices
  - Driver doesn’t have visibility
  - Data access is restricted
Vulkan: Explicit dependencies

- Vulkan has direct support for this type of rendering work load
- By telling the driver how you intend to use the rendered results, the driver can produce a better mapping to the hardware
  - The extra information is a little verbose, but simpler than handling all possible cases yourself!
Vulkan render passes and subpasses

- A render pass groups dependent operations
  - All images written in a render pass are the same size
Vulkan render passes and subpasses

• A render pass groups dependent operations
  - All images written in a render pass are the same size

• A render pass contains a number of subpasses
  - Subpasses describe access to attachments
  - Dependencies can be defined between subpasses
Vulkan render passes and subpasses

- A render pass groups dependent operations
  - All images written in a render pass are the same size
- A render pass contains a number of subpasses
  - Subpasses describe access to attachments
  - Dependencies can be defined between subpasses
- Each render pass instance has to be contained within a single command buffer (unit of work)
  - Some tilers schedule by render pass
Defining a render pass

- VkRenderPassCreateInfo
  - VkAttachmentDescription *pAttachments
    - Just the descriptions, not the actual attachments!
  - VkSubpassDescription *pSubpasses
  - VkSubpassDependency *pDependencies

- vkCreateRenderPass(device, createInfo,.. pass)
  - Gives you a VkRenderPass object
  - This is a template that you can use repeatedly
    - When we use it, we get a render pass instance
Describing attachments for a render pass

- VkAttachmentDescription
  - format/samples
  - loadOp
    - VK_ATTACHMENT_LOAD_OP_LOAD to preserve
    - VK_ATTACHMENT_LOAD_OP_DONT_CARE for overwrites
    - VK_ATTACHMENT_LOAD_OP_CLEAR uniform clears (e.g. Z)
  - storeOp
    - VK_ATTACHMENT_STORE_OP_STORE to output it
    - VK_ATTACHMENT_STORE_OP_DONT_CARE may discard after the render pass
Defining a subpass

- **VkSubpassDescription**
  - `pInputAttachments`
    - Which of the render pass’s attachments this subpass reads
  - `pColorAttachments`
    - Which ones this subpass writes (1:1 - optional)
  - `pResolveAttachments`
    - Which ones this subpass writes (resolving multisampling)
  - `pPreserveAttachments`
    - Which attachments need to persist across this subpass
  - Subpasses are numbered and ordered
Defining subpass dependencies

- **VkSubpassDependency**
  - srcSubpass
  - dstSubpass
    - Where the dependency applies (can be external)
  - srcStageMask
  - dstStageMask
    - Execution dependencies between subpasses
  - srcAccessMask
  - dstAccessMask
    - Memory dependencies between subpasses
Vulkan framebuffers

• A VkFramebuffer defines the set of attachments used by a render pass instance

• VkFramebufferCreateInfo
  - renderPass
  - pAttachments
    - These are actual VkImageViews this time!
  - width
  - height
  - layers
Starting to use a render pass

- **vkCmdBeginRenderPass/vkCmdEndRenderPass**
  - Starts a render pass *instance* in a command buffer
  - You start in the first (maybe only) subpass implicitly
  - pRenderPassBegin contains configuration

- **VkRenderPassBeginInfo**
  - VkRenderPass renderPass
  - The render pass “template”
  - VkFramebuffer framebuffer
  - Specifies targets for rendering
Putting it all together...

Key:
- Objects are dark grey
- Functions are light grey
- Arrows between objects are references of some sort
- Arrows into functions are arguments
- Arrows out of functions are constructed objects
Simple rendering

- `vkAllocateCommandBuffers` *(VK_COMMAND_BUFFER_LEVEL_PRIMARY)*
- `vkBeginCommandBuffer`
- `vkCmdBeginRenderPass`
- `vkCmdDraw` (etc.)
- `vkCmdEndRenderPass`
- `vkEndCommandBuffer`
- `vkQueueSubmit`
Multiple render passes

- You can have more than one render pass in a command buffer
  - Yes, Leeloo multipass, we know...
  - So a command buffer can render to many outputs
    - E.g. you could render to the same shadow and environment maps every frame by reusing the same command buffer
    - But it must be the same outputs each time you submit
    - A specific render pass instance has fixed vkFrameBuffers!
Two limitations...

• **Different render passes ⇒ independent outputs**
  - Rendering goes off-chip, there’s no PLS-style on-chip reuse of pixel contents

• **You can’t reuse the same command buffer with a different render target**
  - E.g. for double buffering or streamed content
  - We’ll come back to this...

• **Still sometimes all you need, though!**
More than one subpass

- `vkCmdNextSubpass` moves to the next subpass
  - Implicitly start in the first subpass of the render pass
  - Dependencies say what you’re accessing from previous subpasses
  - Same render pass so accesses stay on chip (if possible)
Using multiple subpasses

• `vkCmdBeginCommandBuffer`
• `vkCmdBeginRenderPass`
• `vkCmdDraw (etc.)`
• `vkCmdNextSubpass`
• `vkCmdDraw (etc.)`
• `vkCmdEndRenderPass`
• `vkCmdEndCommandBuffer`
Accessing subpass output in fragment shaders

- In SPIR-V, previous subpass content is read with OpImageRead
  - Coordinates are sample-relative, and need to be 0
  - OpTypeImage Dim = SubpassData

- In GLSL (using GL_KHR_vulkan_gls1):
  - Types for subpass access are [ui]subpassInput(MS)
  - layout(input_attachment_index = i, ...) uniform subpassInput t; to select a subpass
  - subpassLoad() to access the pixel

C.f. __pixel_localEXT layouts in EXT_shader_pixel_local_storage when using OpenGL ES

UK Khronos Chapter meet, May 2016
Avoiding unnecessary allocations

• If we’re using subpasses, we likely don’t need the images in memory
  - A tiler may be able to process the subpasses entirely on-chip, without needing an allocation
  - Still need to “do the allocation” in case the tiler can’t handle the request/on an immediate-mode renderer!
    - Won’t commit resources unless it actually needs to

• vkCreateImage flags for “lazy committal”
  - VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT

UK Khronos Chapter meet, May 2016
Vulkan subpasses: advantages

• The driver knows what you’re doing
  - It can reorder subpasses
  - It can change the tile size
  - It can balance resources between subpasses
  - *It will fall back to memory for you* if it has to
  - Under the hood, mechanism likely matches PLS

• Works on immediate mode renderers
  - Probably MRTs and normal external writes
  - Desktop debugging tools will work!

**EXT_shader_pixel_local_storage** is actually more explicit than Vulkan here (and may still be offered as an extension)
There’s more: Secondary command buffers

- Vulkan has two levels of command buffers
  - Determined by vkAllocateCommandBuffers

- **VK_COMMAND_BUFFER_LEVEL_PRIMARY**
  - Main command buffer, as we’ve seen so far

- **VK_COMMAND_BUFFER_LEVEL_SECONDARY**
  - Command buffer that can be invoked from the primary command buffer
Use of secondary command buffers

• **vkBeginCommandBuffer**
  - Takes a VkCommandBufferBeginInfo

• **VkCommandBufferBeginInfo**
  - flags include:
    - VK_COMMANDBUFFER_USAGE_RENDER_PASS_CONTINUE_BIT
  - pInheritanceInfo

• **VkCommandBufferInheritanceInfo**
  - renderPass and subpass
  - framebuffer (can be null, more efficient if known)
Why do we need the “continue bit”?

- Render passes (and subpasses) can’t start in a secondary command buffer
- Non-render pass stuff can be in a secondary buffer
  - You can run a compute shader outside a render pass
- Otherwise, the render pass is inherited from the primary command buffer
Secondary command buffers and passes

• Why specify render pass/framebuffer?
  - Command buffers needs to know this when recording
    - Some operations depends on render pass info (e.g. format)
  - Framebuffer is optional (can just inherit)
    - If you can specify the actual framebuffer, the command buffer can be less generic and therefore may be faster
Invoking the secondary command buffer

- You can’t submit a secondary command buffer
- You have to invoke it from a primary command buffer with `vkCmdExecuteCommands`
Secondary command buffer code

- `vkCmdBeginCommandBuffer`
- `vkCmdBeginRenderPass`
- `vkCmdExecuteCommands`
- `vkCmdNextSubpass`
- `vkCmdExecuteCommands`
- `vkCmdEndRenderPass`
- `vkCmdEndCommandBuffer`
Creating a command buffer can be slow
- Lots of state to check, may require compilation
  - This happens in GLES as well, you just don’t control when!

So create secondary command buffers on different threads
- Lots of 4- and 8-core CPUs in cell phones these days

Invoking the secondary buffer is lightweight
- Primary command buffer generation is quick(er)
What does this have to do with passes?

• Remember:
  - Render passes exist within (primary) command buffers
    - The command buffer sets up the GPU for the render pass
  - On-chip rendering happens within a render pass
    - If you want content to persist between render passes, it’ll reach memory (or at least cache), not stay in the tile buffer
  - You can’t use multiple threads to build work for a primary command buffer in parallel
    - You can build many secondary command buffers at once
You can’t mix and match

- Within a subpass you can either (but not both):
  - Execute rendering commands directly in the primary command buffer
  - `VK_SUBPASS_CONTENTS_INLINE`
You can’t mix and match

- Within a subpass you can either (but not both):
  - Execute rendering commands directly in the primary command buffer
    - VK_SUBPASS_CONTENTS_INLINE
  - Invoke secondary command buffers from the primary command buffer with `vkCmdExecuteCommands`
    - VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS
You can’t mix and match

• Within a subpass you can either (but not both):
  - Execute rendering commands directly in the primary command buffer
    - VK_SUBPASS_CONTENTS_INLINE
  - Invoke secondary command buffers from the primary command buffer with vkCmdExecuteCommands
    - VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS
  - Chosen by vkCmdBeginRenderPass/vkCmdNextSubpass
    - Remember: you can only do these in a primary command buffer!
Command buffer reuse: even faster

• Primary command buffers work with a fixed render pass and framebuffer
  - You can reuse a primary command buffer, but it will always access the same images - often good enough
    - May have to wait for execution to end; can’t be “one-time”

• What if you want to access different targets?
  - E.g. a cycle of framebuffers or streamed content?
  - You can round-robin several command buffers
  - Or you can use secondary command buffers!
Compatible render passes and frame buffers

• The render pass a secondary command buffer uses needn’t be the one it was recorded with:
  - It can be “compatible”
    - Same formats, number of sub-passes, etc.

• You can have primary command buffers with different outputs, and they can re-use secondary command buffers:
  - The primary has to be different to record new targets
  - The primary may have to patch secondary addresses
Almost-free use with changing framebuffers

• No cost for secondary command buffers
• Primary command buffer is simple and quick
So I can do bloom/DoF/rain/motion blur…!

• No! Remember, you can only access the current pixel

• Tilers process one tile at a time
  - If you could try to access a different pixel, the tile containing it may not be there
  - You have to write out the whole image to do this
    - Slow, painful, last resort!
  - Yes, we can think of possible solutions too
    - Give it time (lots of different hardware out there)
Coming out of the shadow(buffer)s

• Render passes are integral to the Vulkan API
  - Reflects modern, high-quality rendering approaches

• The driver has more information to work with
  - It can do more for you
    - Remember this if you complain it’s verbose!

• Hardware resource management is hard
  - Expect drivers to get better over time

• Another tool for better mobile gaming
Thank you

• Over to you...

Andrew Garrard
a.garrard at samsung.com