Vulkanised 2019
Live Long and Optimise

Michael Parkin-White, Calum Shields
m.parkin-whi@samsung.com  c.shields@samsung.com
Intro – Samsung Galaxy GameDev

Promoting use of Vulkan on Android
Support studios remotely & on-site
Help partners optimise their games through use of tools and best-practices
Optimal Swapchain Management

An in-depth investigation - beyond the basics
Vulkan Swapchain & presentation

- Consists of three interactions w/ presentation engine:
  - `vkAcquireNextImageKHR(..)` – Potential blocking if no swapchain image is available
  - `vkQueueSubmit(..)` – Non blocking
  - `vkQueuePresentKHR(..)` – Non blocking
  - `vkQueuePresentKHR(..)` – Frequently blocks on Android - in some cases for > 20ms!
Standard swapchain coordination

- A standard approach to swapchain management:

- However, we have problems when blocking behaviour emerges – increased CPU time on main Vulkan thread:
Consideration – Delayed Acquire

- In games with highly variable frame timings, there is a benefit to having two points at which we attempt `vkAcquireNextImageKHR`. 

![Diagram showing the process of command recording, submitting an image, and delays in acquiring the image before presenting it.]
The blocking vkQueuePresentKHR issue

- We have observed that vkQueuePresentKHR can block for significant durations on Android

`queueBuffer` (called by `vkQueuePresentKHR`) taking an average of 12ms
Solution #1 – Delayed Present

- This delay is influenced by proximity to vkQueueSubmit – So we can instead delay the call to present.
Results – Delayed Present

**Default Present:** queueBuffer takes an average of **12ms**

**Delayed Present:** queueBuffer takes an average of **0.2ms**
Solution #2 – Presentation Thread

- Another solution is to defer swapchain interaction calls to a separate thread:
  - In this case, we move the calls off the work/recording thread – allowing the wait to be absorbed externally
  - We can continue with useful CPU work on the main thread.
    - A synchronisation check should be added to prevent the CPU from getting too many frames ahead (i.e. more than 2 ahead).
Results – presentation thread

- With the presentation thread implemented in **UE4 Sun-temple** demo:

<table>
<thead>
<tr>
<th></th>
<th>Standard UE4.22.0 VulkanRHI</th>
<th>UE4.22.0 VulkanRHI With presentation thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>S960U – Locked Frequency</td>
<td>FPS: 34</td>
<td>FPS: 41</td>
</tr>
</tbody>
</table>

Thread-2 is now used for presentation

RHI Thread is now free to get on with useful work!
Pipeline Analysis
Optimising rendering workflow
Example: Pipeline analysis

Poor pipelining results in pipeline bubbles – Not getting the most out of GPU

Fragment and vertex work never running in parallel

Caused by sub-optimal pipeline barrier and subpass-dependency stage masks

Current Frame Cost: 62.5ms (16 fps!)
Galaxy GameDev

Pipeline Barriers: Quick overview

- Used to specify **execution dependencies** between specific pipeline stages in two action commands.

- **Destination stage mask**: Specifies where the 2nd (next) action item will wait for the 1st (previous) action to complete its **Source stage mask** stages.
Pipeline barriers example

- **Simplified Example**: Render with two passes. Shadow mapping and main render

- Main scene render needs to use the shadow map rendered in the first pass

- Naïve synchronisation assumes entire shadow pass needs to complete **before** we start the main scene’s rendering work

```
srcStageMask: VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
dstStageMask: VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
```
Pipeline Barriers: Improved case!

Source stage = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

Destination stage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
Pipeline barriers example

- **Optimal case**: We can modify the stage masks and allow the main scene vertex work to overlap the shadow-pass fragment work.

- Results in **6ms** saving! ALU-dependent vertex operations can run in parallel with Texture-dependent fragment operations.
Frame-time: 40ms - **56% Performance increase** - with one line of code!

**Before** – General pipeline barriers

**After** – Optimal per-pass pipeline barriers
Further Pipeline Optimization – Removing Render Passes

- **High Vertex Load** – Vertex work expensive on tile-based GPUs
- **AAA Engine ported to Android from PC/Console** – Non-optimal for mobile HW

![Standard Trace](chart1)

- **Depth Pre-pass**
- **Occlusion Queries**

**Removing Depth pre-pass**

Potential performance: **35 fps**!

(62.5ms -> 28ms/frame)
Subpasses

Optimising rendering for memory bandwidth
Useful Vulkan Features - Subpasses

- Allows efficient performing of additional render workload where on-tile frame contents are preserved
  - Large bandwidth savings
  - Avoid GPU Idle time spent storing and loading framebuffer data to main memory
  - Potential power saving and performance increases
  - Use of transient attachments and lazy memory allocation
Subpass Viability

- Shaders in next render pass only sample local framebuffer data
- Next render pass uses the same framebuffer attachments
- Material nodes e.g. **PixelDepth, SceneDepth** and **SceneColor** *could* imply subpass compatibility!

**Note:** Many UE4 “Translucency-pass” shaders only sample the local depth value!

**Note:** If we need sparse sampling of a framebuffer, we cannot benefit from subpasses.

These are perfect candidates for Subpass optimisation.
Generic Shader Compatibility test

- Test sample coordinate against local pixel coordinate – using RenderDoc

\[
\text{vec2 depth_sample_uv} = ((v5.xy / v5.ww) * _18.pu_h[12].xy) + _18.pu_h[12].wz;
\]

\[
\text{float DIFF} = \text{depth_sample_uv} - \text{vec2(gl_FragCoord.x / 1376.0, gl_FragCoord.y / 720.0)};
\]

\[
\text{gl_FragColor.rgb} = 1.0 - \text{vec3(DIFF)};
\]

Allows us to determine whether depth sample in “Translucency pass” is local

In this case, god ray shader is subpass compatible!
Depth, colour and stencil targets stored and re-loaded

Standard UE4 Render pass

GPU spends \(~1.0\text{ms}\) idle

Using Subpasses

Depth, colour and stencil remain on-tile

GPU spends \(~0.15\text{ms}\) idle

Bandwidth saving of 700MB/s

Reduced Fragment Idle Time by \(~1\text{ms}\)!
Subpass Performance Results

<table>
<thead>
<tr>
<th></th>
<th>FPS</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default – No additional Subpasses</td>
<td>52</td>
<td>11.1%</td>
<td>97%</td>
</tr>
<tr>
<td>Using Subpasses (No Depth and Color load/store)</td>
<td>55</td>
<td>12.4%</td>
<td>97%</td>
</tr>
</tbody>
</table>

~6% Performance increase in GPU fragment-bound use case on Galaxy S9!
Vulkan Tips and Tricks

Quick points for optimisation
Load and Store Appropriately

- **LOAD_OP_LOAD** will read the attachment data in system memory into the tile buffer
  - Costs a lot of bandwidth

- **LOAD_OP_CLEAR & LOAD_OP_DONT_CARE** will set the clear value in the tile buffer directly
  - Costs no bandwidth

- **STORE_OP_STORE** will write the attachment back out to system memory
  - Costs a lot of bandwidth

- **STORE_OP_DONT_CARE** writes nothing out
  - Costs no bandwidth
Clear Efficiently

- Don’t ever use vkCmdClearColorImage or vkCmdClearDepthStencilImage!!
  - Wastes Bandwidth unnecessarily
  - Use loadOp Clear at beginning of renderpass
  - Use vkCmdClearAttachments mid renderpass

Optimal Clear 62.7MB/s

Unoptimal Clear 1.7GB/s
Transient Attachments

- Attachments that exist solely in tile memory
  - Doesn’t need to be backed by memory
  - Reduces memory footprint

- Required flags
  - `imageUsage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`
  - `memoryProperty = VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT`
Upload buffer data to GPU

- No need to use staging buffers for copying CPU buffer data to GPU
- UMA on mobile devices
- Still required for uploading image data to GPU
Tiling (of images)

- Raster order doesn’t usually suit textures
- TILING_LINEAR is useful for frequent updates
- Use TILING_OPTIMAL for better GPU cache access
New Vulkan Feature – Depth Stencil Resolve

- MSAA is cheap on mobile tile-based architectures
- Resolving MSAA depth targets is currently expensive
- New Vulkan Extension to enable efficient on-tile resolve

What this Means:
- Depth-dependent Renderpass effects such as:
  
  - Translucency pass - decal projection - depth of field - god rays - fog

- Possible with MSAA enabled at no additional performance cost!
Android Tools
Best tools for the job
Tools - RenderDoc

Full static frame analysis.

Verify API usage:
- Draw calls
- Renderpasses
- Barriers
- Resources

Step-through scene

Informed content optimisation!

Works very well with Vulkan!
Tools – Arm Streamline, Snapdragon Profiler

In-depth hardware analysis

Counters:
- Vertex Activity
- Fragment Activity
- CPU core utilisation
- Memory analysis

High-resolution data

Identify bottlenecks!

Visually analyse render workload execution

Improve app performance with high-quality Vulkan use
GPU Watch

- Performance monitoring tool
  - Direct result on the screen
  - Support Vulkan/OpenGL ES
Thank You

• Correct Pipeline Barrier staging
• Use Subpasses where you can
• Load & Store Appropriately

• Use Transient Attachments
• Clear Efficiently