Vulkan: Mark My DWORDS

Hai Nguyen / Google

Vancouver, August 2018

Agenda

- Overview
- Getting Started
- Marking Buffers
- Viewing Markers
- How Is This Useful?
- Practical Concerns
- Practical Experience
- Turning It Up to 11
- We’re Hiring

Overview

FADE IN:

INT. PROGRAMMER OFFICE - DESK - DAY,NIGHT (HARD TO TELL - IT'S PROGRAMMING)

Programmer sits at desk typing away furiously. Programmer hits compile. Compile is successful. Programmer hits run.

CUT TO MONITOR DURING PROGRAM STARTUP

Splash screen good. Level loads. First frame renders.

CUT TO WASD + MOUSE INPUT
Programmer starts to test.

CUT BACK TO MONITOR AT PROGRAM CRASH

Debugger has asserted at vkQueueSubmit. Return code is VK_ERROR_DEVICE_LOST.

WIDE TO BEHIND PROGRAMMER

Programmer raises arms in the air with clenched fist.

PROGRAMMER

What the DWORD just happened? What crashed on the GPU? Which command buffer was it?! Can't you just tell me, something?! Anything?! (SOFT SOBBING SOUNDS)

Overview

- Show how to use **VK_AMD_buffer_marker** to track command buffer progress to bubble possible source of crashes for debugging
  - Possible general version of extensions available in the future
- Supplies you'll need for trying this at home
  - AMD GPU with Vulkan driver support for **VK_AMD_buffer_marker**
  - *Note: I haven't tested any of this on RADV. Don't think the extension is available there.*

Overview

- Text in this presentation is slightly color coded
  - I’m doing this to distract you because I don’t have any fancy pictures :(
  - Hypothetical audience questions are quoted in *italic* purple
    - *“Are you serious?”*
    - Yes
- Code is syntax highlighted for easier reading

```
// Print something to console
std::cout << “Hello, Khronos BoF!” << std::endl;
```

- Some of the Vulkan constants are shortened
  - Because they’re really really long

Getting Started

- *"Do I need any extensions?"*
  - Yes: **VK_AMD_buffer_marker**
  - It's a VkDevice extension
- *"What's structs come with VK_AMD_buffer_marker?"*
  - None
- *"What functions come with VK_AMD_buffer_marker?"*
  - **vkCmdWriteBufferMarkerAMD**
- *"What are the requirements of the VkBuffer object?"*
  - No specific memory restrictions
  - Can simply be a **HOST_VISIBLE** buffer

Marking Buffers

- The One Function:

```
void vkCmdWriteBufferMarkerAMD(
    /* The command buffer into which the command will be recorded */
    VkCommandBuffer        commandBuffer,

    /* The pipeline stage whose completion triggers the marker write */
    VkPipelineStageFlagBits pipelineStage,

    /* The buffer where the marker will be written to */
    VkBuffer               dstBuffer,

    /* The byte offset into the buffer where the marker will be written to */
    VkDeviceSize           dstOffset,

    /* The 32-bit value of the marker, AKA the DWORD! */
    uint32_t               marker
);
```

Marking Buffers

- *"Wait, where does this 'marking the buffer' happen?"*
- From the spec

  *"The command [**vkCmdWriteBufferMarkerAMD**] will write the 32-bit marker value into the buffer only after all preceding commands have finished executing up to at least the specified pipeline stage."*
- What this means
  - **vkCmdWriteBufferMarkerAMD** must be placed after the command (e.g. **vkCmdDraw**, **vkCmdDrawIndexed**, etc.)
  - If the **pipelineStage** is **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT** it will write the specified marker value after this stage is done executing
  - If something happens before or during **pipelineStage**, then no writes occur

Marking Buffers

- The One Function:

```
VkDeviceSize offset = 0;
// Draw 1
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   buf, offset,     1);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, buf, offset + 4, 2);
// Draw 2
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   buf, offset + 8,  3);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, buf, offset + 12, 4);
// Draw 3
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   buf, offset + 16, 5);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, buf, offset + 20, 6);
```

Contents of **buf**

Byte Offset	Value
0	1
4	2
8	3
12	4
16	5
20	6

Viewing Markers

- If you have a host visible marker buffer:

```
// Print markers to console
for (uint32_t i = 0; i < marker_count; ++i) {
  uint32_t offset = 4 * i;
  uint32_t value = *(mapped_address + offset);
  std::cout << "Marker " << i << " : " << value << "\n";
}
```

- Host visible marker buffers can be viewed at anytime
- Practically speaking, **VK_ERROR_DEVICE_LOST** is the best time to dump marker buffers
  - E.g. Dump marker buffer to console when **vkQueueSubmit** returns **VK_ERROR_DEVICE_LOST**

Useful How?

- *“Cool, so I can get the GPU to write a DWORD from the command buffer as it’s finishing up executing things...how exactly is this useful?”*

How Is This Useful?

- Tracking shader crashes

```
// Zero out marker buffer
SetBufferMarkerToZero(bufN);

// Record Command buffer
VkDeviceSize offset = 0;
// Bind pipeline1 and draw
vkCmdDraw(...);
vkCmdBindGraphics(cmd, GRAPHICS, pipeline1);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset,     1);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 4, 2);
/ Bind pipeline2 and draw
vkCmdBindGraphics(cmd, GRAPHICS, pipeline2);
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset + 8,  3);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 12, 4);
/ Bind pipeline3 and draw
vkCmdBindGraphics(cmd, GRAPHICS, pipeline3);
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset + 16, 5);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 20, 6);
```

Contents of **bufN**

Byte Offset	**Value**
0	**0**
4	**0**
8	**0**
12	**0**
16	**0**
20	**0**

How Is This Useful?

- Tracking shader crashes
  - Command buffer for frame N record and submitted
  - Move onto frame N+1
  - Command buffer for frame N+1 record and submitted…
  - vkQueueSubmit returns **VK_ERROR_DEVICE_LOST**
  - Dump marker buffer **bufN**
    - Only first 3 entries are populated
    - Something happened in **pipeline2’s** fragment shader

```
// Zero out marker buffer
SetBufferMarkerToZero(bufN);
// Record Command buffer
vkDeviceSize offset = 0;
// Bind pipeline1 and draw
vkCmdDraw(...);
vkCmdBindGraphics(cmd, GRAPHICS, pipeline1);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset,     1);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 4, 2);
/ Bind pipeline2 and draw
vkCmdBindGraphics(cmd, GRAPHICS, pipeline2);
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset + 8,  3);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 12, 4);
/ Bind pipeline3 and draw
vkCmdBindGraphics(cmd, GRAPHICS, pipeline3);
vkCmdDraw(...);
vkCmdWriteBufferMarkerAMD(cmd,VERTEX_SHADER,   bufN, offset + 16, 5);
vkCmdWriteBufferMarkerAMD(cmd,FRAGMENT_SHADER, bufN, offset + 20, 6);
```

Contents of **bufN**

Byte Offset	**Value**
0	1
4	2
8	3
12	**0**
16	**0**
20	**0**

How Is This Useful?

- In real world scenarios..
  - Possibly hundreds or thousands of pipelines and shader permutations
  - Possibly 10, 20, or even 100 command buffers in flight at once
  - Crash can happen on random a frame
  - Crash can happen at random places in a level or scene
  - If the crash is coming from a shader, this is where **VK_AMD_buffer_marker** helps
- Marker values
  - Encoding scheme for marker values
    - For example if you want to track shader stage progress:

```
uint32_t value = 10*commandNumber + VK_SHADER_STAGE_*_BIT;
```

  - Start/End marker values
    - Makes things easier when looking for incomplete command buffers

Practical Concerns

- Pairing Command Buffer and Marker Buffer
  - 1:1 pairing of marker buffers to in-use command buffers
  - Avoid having in-use command buffers share same marker buffer
    - Gets really confusing fast
- Allocation Limits
  - Suballocate memory for buffers to avoid running into device allocation limits
  - Most drivers for AMD GPUs impose a 4096 allocation count limit
- Performance
  - Slight overhead in writing to marker buffer
- Zeroing out buffer
  - Consider zeroing out marker buffer as opposed to just overwriting

Practical Concerns

- **VK_AMD_buffer_marker** is handy for tracking down crashy shaders
  - Not a magic bullet
  - Developer still required to debug crashy shaders
  - Makes locating crash source easier
  - Reduces the need to comment out code until crashing stops
- Caveats
  - **VK_ERROR_DEVICE_LOST** is the most useful error to stop on
  - Other error messages may produce mixed results
    - Marker buffers may be all zeros or fully written
    - Very least tells you which command buffers completed before error
  - Limited to AMD GPUs

Practical Experience

- Working on a Vulkan demo
  - Next version of a “Fish Tornado” style demo
  - Had a random crash that couldn’t easily pin down
  - Added VK_AMD_buffer_marker to renderer
    - Took about ~2 hours, ran into allocation limits, implemented suballocation
  - Found source of crash about 10 minutes later
    - Shader in global illumination lighting pass
    - TL;DR; bad offsets in constant buffer caused a chain of bad accesses
- Overhead of buffer marking
  - Cost was about ~3fps
  - Totally forgot it was running after fixing issue

Turning It Up to 11

- Instead of embedding in app...write a layer
  - VK_AMD_buffer_marker + Layer Factory
  - Use this layer with any of your Vulkan applications
- Combine with SPIR-V reflection for even more info
  - If SPIR-V is built with debug info, print out the shader source file name
  - Requires bookkeeping to track pipelines and shader modules
- Combine with **VK_AMD_shader_info** for GCN
  - Save GCN during pipeline creation
    - Print to file or console when **VK_ERROR_DEVICE_LOST** occurs
  - Also requires additional bookkeeping

We're Hiring!

Contact Kevin Lusby

(kevinlusby@google.com)!

Come visit us at the Google booth!