Using Vulkan Synchronization Validation Effectively

John Zulauf, LunarG

Presentation:
https://bit.ly/3U5PtWU
Why Synchronization Validation?

- **Vulkan Synchronization Is Challenging**
  - Massively parallel implementations, few ordering guarantees
  - Robust, complex synchronization capabilities in Vulkan API
  - Performance implications of *too much* synchronization
  - Need ensure *correctness*, not just correct *appearance*

- **Quick Level Set**
  - Technical deep-dive into using Synchronization Validation to find and debug issues
  - Assumes working knowledge of Vulkan Synchronization functionality
Synchronization Validation

- Detects Hazard From Insufficient Synchronization Operations
  - Hazard -- any access were the access pattern is not well defined
  - Byte Resolution Access/Synchronization Tracking
  - All vkCmd types (transfer, draw, renderpass, compute, resolve, etc)
  - Sync2 support

- Inter-Command Buffer Support
  - vkCmdExecuteCommands
  - Queue Submit
  - Binary Semaphores
  - Fence
  - Queue|Device Wait Idle
Synchronization Validation Limitations

- Limited aliasing detection (like kinds of resources)
- No timeline semaphore support
- No Host side resource tracking
- No swizzle support
- Not GPU Assisted (doesn’t know shader execution time information)
- Limited extension support
- Challenging to use
Using Synchronization Validation

● **Clean Validation Run**
  ○ Resolve all outstanding non-synchronization issues.
  ○ Recommend “GPU Assisted” as well.

● **Running**
  ○ Enable Synchronization Validation (next slide)
  ○ Disable all other validation
  ○ Chase down issues in debugger.
    ■ “Debug Action: Break” on Windows
    ■ Break in vkCreateDebugUtilsMessengerEXT callback
Enabling Synchronization Validation

- **Vkconfig**
  - Select the “Synchronization Preset”

- **vk_layer_settings.txt**

```plaintext
khrnos_validation.enables = VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT
Khronos_validation.disables =
  VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT, VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT, VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT
```

- **Environment variables**

```plaintext
VK_LAYER_ENABLES=VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT
VK_LAYER_DISABLES=VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT;VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT;VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT
```
“Congratulations! It’s an Error”

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0xfa21a40000000003, type = VK_OBJECT_TYPE_BUFFER; | MessageID = 0x376bc9df | vkCmdCopyBuffer(): Hazard WRITE_AFTER_READ for dstBuffer VkBuffer 0xfa21a40000000003[], region 0. Access info (usage: SYNC_COPY_TRANSFER_WRITE, prior_usage: SYNC_COPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), command: vkCmdCopyBuffer, seq_no: 1, reset_no: 1).

● Now what?
  ○ Step 1: Understanding Hazard Messages
  ○ Step 2: Finding the Missing Synchronization

● But first some background…
Synchronization Validation Operations

- Tracks access history
  - Operation Type as Stage/Access pairs
  - Stores “first” and “most recent” prior only
- Applies synchronization operations to access history
  - Identifies “safe” subsequent access operations
  - Track dependency chaining
- Validates accesses of each operation against prior accesses
  - The stage and access for each are compared prior access and synchronization
  - Reports hazards
  - Any hazard reported earlier may mask detection of subsequent hazard with same memory
Synchronization Validation Concepts

- **Stage/Access pairs**
  - Describe the usage of resources
  - Not all pairs are valid, valid pairs expressed as enum `SYNC_<STAGE>_<ACCESS>`
  - Meta stages/access for non-pipeline operations (e.g. layout transition)

- **“Prior”, “Current”, and “First”**
  - Hazard reports always reference two stage/access usages (prior and current/first)
  - Relative to a specific resource
  - Barrier information reflects synchronization operations between “prior” and “current/first”

- **Access Operations**
  - Commands that access (or record operations that will modify) resources

- **Synchronization Operations**
  - Commands that enforce (or record operations that will enforce) ordering between accesses
Record Time vs. Submit Time Validation

- **Record Time**
  - Validates effect of current `vkCmd...` relative to earlier commands in same command buffer
  - `vkCmdExecuteCommands` special; validates effect of “first” access of secondary command buffers
  - Does not validate against any other command buffer

- **Submit Time**
  - Validates effect of “first” access of each submitted command buffer relative to all others in “Queue Submission Order” same queue
  - Validates against all other queue’s submissions including the presence(or absence) of semaphore, wait, and fence operations
Prior, Current, and First Accesses

- "Prior" – most recent access…
  - In command buffer record and submission order (see Queue Submission Order)
  - Most recent non-recorded access in API calling sequence
- "Current"
  - The immediate effect of a command at record time
  - "usage" – For the currently recorded vkCmd... command
- "First"
  - The earliest (in Queue Submission Order) effect of a recorded command
    - Zero or more reads
    - Zero or one write
  - "executed_usage" – The first access of executed command buffer
  - "submitted_usage" – The first access submitted command buffer
## Types of synchronization errors

<table>
<thead>
<tr>
<th>RAW</th>
<th>Read-after-write</th>
<th>This occurs when a subsequent operation uses the result of a previous operation without waiting for the result to be completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAR</td>
<td>Write-after-read</td>
<td>This occurs when a subsequent operation overwrites a memory location read by a previous operation before that operation is complete. (requires only execution dependency)</td>
</tr>
<tr>
<td>WAW</td>
<td>Write-after-write</td>
<td>This occurs when a subsequent operation writes to the same set of memory locations (in whole or in part) being written by a previous operation</td>
</tr>
<tr>
<td>WRW</td>
<td>Write-racing-write</td>
<td>This occurs when unsynchronized subpasses/queues perform writes to the same set of memory locations</td>
</tr>
<tr>
<td>RRW</td>
<td>Read-racing-write</td>
<td>This occurs when unsynchronized subpasses/queues perform read and write operations on the same set of memory locations</td>
</tr>
</tbody>
</table>
Synchronization Validation Operations (revisited)

- Tracks access history
  - How does the current operation (draw, transfer, etc.) affect the resource
  - Stage/access of operation for each resource
  - Include implicit operations (layout transition, load, resolve, store)
  - “First” access of an executed or submitted command buffer

- Applies synchronization operations
  - What relation do synch operations have relative to a given resource?
  - Do they apply at all? Also include earlier synch operations (chaining)
  - What subsequent operations are “safed” for that resource

- Validates accesses of each operation against prior accesses
  - What are the prior commands that touch a given resource (memory location)?
  - Comparison to earlier command stage/access and sync operations (“..is it safe?”)
  - Command from earlier queue submissions
  - Accesses from acquire or present
Step 1: Understanding Hazard Messages

- Lots of information
- Densely Packed
Record Time Hazard

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0xfa21a4000000003, type = VK_OBJECT_TYPE_BUFFER; MessageID = 0x376bc9df

| vkCmdCopyBuffer(): Hazard WRITE_AFTER_READ for dstBuffer VkBuffer 0xfa21a4000000003[], region 0. Access info (usage: SYNC_COPY_TRANSFER_WRITE, prior_usage: SYNC_COPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), command: vkCmdCopyBuffer, seq_no: 1, reset_no: 1).

vkCmdCopyBuffer() is the current command being recorded
Record Time Hazard (cont’d)

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0xfa21a40000000003, type = VK_OBJECT_TYPE_BUFFER; | MessageID = 0x376bc9df
| vkCmdCopyBuffer(): Hazard WRITE_AFTER_READ for dstBuffer VkBuffer 0xfa21a40000000003[], region 0. Access info (usage: SYNC_COPY_TRANSFER_WRITE, prior_usage: SYNC_COPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), command: vkCmdCopyBuffer, seq_no: 1, reset_no: 1).

usage – vkCmdCopyBuffer is writing to the destination buffer at the transfer stage
prior_usage – the most recent previous access was a read at the transfer stage
Record Time Hazard (cont’d)

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0xfa21a40000000003, type = VK_OBJECT_TYPE_BUFFER; | MessageID = 0x376bc9df

vkCmdCopyBuffer(): Hazard WRITE_AFTER_READ for dstBuffer VkBuffer 0xfa21a40000000003[], region 0. Access info (usage: SYNCHCOPY_TRANSFER_WRITE, prior_usage: SYNCHCOPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), command: vkCmdCopyBuffer, seq_no: 1, reset_no: 1).

command – vkCmdCopyBuffer was the command that read from the buffer

read_barriers – there are no barriers to read operations since prior_usage

seq_no and reset_no – indicate the where in the command buffer the read lives

17
Submitted Command Buffer Hazard

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0x1febb508d20, type = VK_OBJECT_TYPE_QUEUE; | MessageID = 0x376bc9df | vkQueueSubmit(): Hazard WRITE_AFTER_READ for entry 1, VkCommandBuffer 0x1febae67c50[], Submitted access info (submitted_usage: SYNC_COPY_TRANSFER_WRITE, command: vkCmdCopyBuffer, seq_no: 1, reset_no: 2). Access info (prior_usage: SYNC_COPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), queue: VkQueue 0x1febb508d20[], submit: 0, batch: 0, batch_tag: 1, command: vkCmdCopyBuffer, command_buffer: VkCommandBuffer 0x1fec5015920[], seq_no: 1, reset_no: 2).

vkQueueSubmit – Submit of command buffer 0x1febae67c50 on queue handle

submitted_usage – Is the first usage within 0x1febae67c50 of the affected resource
Submitted Command Buffer Hazard

Validation Error: [SYNC-HAZARD-WRITE-AFTER-READ] Object 0: handle = 0x1febb508d20, type = VK_OBJECT_TYPE_QUEUE; | MessageID = 0x376bc9df | vkQueueSubmit(): Hazard WRITE_AFTER_READ for entry 1, VkCommandBuffer 0x1febae67c50[], Submitted access info (submitted_usage: SYNC_COPY_TRANSFER_WRITE, command: vkCmdCopyBuffer, seq_no: 1, reset_no: 2). Access info (prior_usage: SYNC_COPY_TRANSFER_READ, read_barriers: VkPipelineStageFlags2(0), queue: VkQueue 0x1febb508d20[], submit: 0, batch: 0, batch_tag: 1, command: vkCmdCopyBuffer, command_buffer: VkCommandBuffer 0x1fec5015920[], seq_no: 1, reset_no: 2).

prior_usage – Information for command_buffer submitted on queue

command – Is the most recent access within command_buffer of the affected resource
Command Type Specific Error Details

- **Copy**
  - Source/Destination
  - Region index

- **Draw or dispatch**
  - Descriptor: binding, type
  - Attachment: index and type
  - Bound buffer: vertex or index

- **Image Barriers**
  - Transitions: oldLayout, newLayout
  - Image Subresource

- **Render pass**
  - Transitions: oldLayout, newLayout
  - load/store/resolve: attachment index, type, and operation
Call To Action 1

Tell us how to improve hazard messages.

Be specific. Give use cases.

Open Github Issue. Link below.
Step 2: Finding the Missing Synchronization

- Frequently Found Issues
- Debugging Using Access info information
- Method of Bisection Using Additional Barriers
- Identifying Affected Resources and Operations
- Using Code Inspection
Frequently Found Issues

- Assuming pipeline stages are logically extended with respect to memory access barriers. Specifying the vertex shader stage in a barrier will not apply to all subsequent shader stages read/write access.
- Invalid stage/access pairs (specifying a pipeline stage for which a given access is not valid) that yield no barrier.
- Relying on implicit subpass dependencies with VK_SUBPASS_EXTERNAL when memory barriers are needed.
- Missing memory dependencies with Image Layout Transitions from pipeline barrier or renderpass Begin/Next/End operations.
- Missing stage/access scopes for load and store operations, noting that color and depth/stencil are done by different stage/access pairs.
Debugging Using Access info information

- **Hazards from Missing or Incomplete Barriers**
  - Zero (empty) Read and Write Barriers – missing barrier or scope
  - Non-Zero Barriers – scope vs. usage mismatch

- **Hazards vs. Prior Image Layout Transitions**
  - Find the last layout transition (barrier or subpass dependency)
  - Usually a missing dstStageMask or dstAccessMask

- **Hazards at Image Layout Transitions**
  - Missing srcStageMask or srcAccessMask for the affected resource

- **Hazards between buffer and/or image resource uses**
  - Write-target to/from Read-target (pre/post transfer, attachment-to/from-texture)
  - Application needs to track the changing roles of a resource
  - Look for where these role changes happen, and check the synchronization operations
Hazards from Missing or Incomplete Barriers

- **Zero (empty) Read and Write Barriers (one of)**
  - Barrier of apropos type was not issued
  - Resource not included in barrier
    - Resource handle not specified in BufferMemoryBarrier/ImageMemoryBarrier
    - Resource usage not included correctly included in barrier *first* (or source) scope

- **Non-Zero Barriers**
  - Barrier affecting resource *has* been used
  - *Current* usage not include in barrier *second* (or destination) scope
Method of Bisection

- Insert “big hammer” Barriers/Subpass Dependency
  - Stage:
    - Outside Renderpass: VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
    - Inside Renderpass: VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
  - Access
    - VK_ACCESS_MEMORY_READ_BIT | VK_ACCESS_MEMORY_WRITE_BIT
- If error disappears, error source is prior to Barrier, else it is after
- Move barrier to determine source of hazard
- Alternatively “Big Hammer” Semaphore or Fence between Queue Submits instead of barrier
- Be sure to remove after – they will impact performance
Identifying Affected Resources and Operations

● Getting Consistent Resource Identification
  ○ Resource handles are not guaranteed to be invariant
  ○ Use `vkSetDebugUtilsObjectNameEXT` and `vkSetDebugUtilsObjectTagEXT`
  ○ Object Names will be shown in hazard messages

● Tracking Operations For a Given Resource
  ○ Use the object name to identify the current handle at `vkSetDebug…` time
  ○ Break at API where handle is referenced and call matches `prior_usage` and `command`
  ○ Note that handle may be referenced indirectly (descriptors, `vkSet…Buffer`, etc)
Region Labels (WIP)

On main branch (and next SDK) VK_EXT_debug_utils support for

\texttt{vkCmdBeginDebugUtilsLabelEXT} \textbf{and} \texttt{vkCmdEndDebugUtilsLabelEXT}

Validation Error: [ \texttt{SYNC-HAZARD-WRITE-AFTER-READ} ] Object 0: handle = 0xfa21a40000000003, type = VK_OBJECT_TYPE_BUFFER; | MessageID = 0x376bc9df |
vkCmdCopyBuffer(): Hazard WRITE_AFTER_READ for dstBuffer VkBuffer 0xfa21a40000000003[], region 0. Access info (usage: \texttt{SYNC_COPY_TRANSFER_WRITE}, prior_usage: \texttt{SYNC_COPY_TRANSFER_READ}, read_barriers: VkPipelineStageFlags2(0), command: vkCmdCopyBuffer, seq_no: 1, reset_no: 1, debug_region: RegionA::RegionB).

define debug_region\textbf{ is the region set current at} prior_usage joined with `::`
Using Code Inspection

- Look near the stack trace location
  - Often missing/malformed barrier is on or near the current stack trace
  - Use the “Zero” and “Non-zero” barrier inspection rules above to evaluate

- Identifying Incomplete Existing Barriers
  - Search the code for VK_PIPELINE_STAGE_* or VK_ACCESS_* matching:
    - The current usage (check dst*Mask) or
    - Prior usage (check src*Mask fields) and
    - Do not include the correct flags for the opposite usage.
Using Code Inspection (cont’d)

- Examining Resource Use Transitions
  - Applications frequently track the *logical* use (or role) of a resource in metadata.
    - E.g. texture vs. rendering target
  - Inspect code which implements the role change
    - Frequently there will be calls to barrier, layout, or queue family ownership calls
    - Inspect these relative to the “Missing or Incomplete Barriers” discussion above
  - Look at objects where the *logical* use mismatches the actual use
    - This may indicate that, while the correct transition code exists, it isn’t being called
Call To Action 2

Tell us what debugging features are missing and needed.

Be specific. Give use cases.

Open Github Issue. Link below.
Two Final Thoughts…

● Be sure and check Core/Parameter Validation as you change code to address synchronization issues.

● Remember that “no corruption” doesn’t imply “correct”
  ○ Timing is implementation specific
  ○ “Be lucky” isn’t a strategy
Today's Presentation:
https://bit.ly/3U5PtWU

Get A FREE Tumbler at the LunarG Sponsor Table!

Thank you!

QUESTIONS?