Ensure Correct Vulkan Synchronization by Using Synchronization Validation

Jeremy Gebben, John Zulauf
LunarG, Inc
August 2021

- Revised February 2022

LUNARG

Understanding Vulkan Synchronization

Jeremy Gebben, LunarG Inc.

Senior Graphics Software Engineer with 25 years of experience working on drivers for GPUs, high speed networking devices, and custom embedded hardware.

Slides are available at:
https://www.lunarg.com/news-insights/white-papers/vulkan-synchronization-siggraph-2021/

Introduction

- Understanding Vulkan Synchronization
  - Synchronization terminology in the Vulkan spec
  - Some of the new features in VK_KHR_synchronization2 (aka sync2)
- Validating Vulkan Synchronization
  - Theory of operation
  - Demo
  - Interpreting error messages
  - Frequently found errors
- Q & A

LUNARG

Execution Dependencies

- Most Vulkan commands are started in queue submission order but may execute in any order
  - Even commands using the same pipeline stages!
- The programmer must tell Vulkan when 2 commands depend on each other
  - We do this by defining barriers
- *First synchronization scope* is what *happens before* a barrier
  - AKA: **srcStageMask**
- *Second synchronization scope* is what *happens after* a barrier
  - AKA: **dstStageMask**

LUNARG

Memory Dependencies

- GPUs have lots of caches
  - Vulkan defines logical memory access types that correspond to caches that *might* be associated with a pipeline stage
  - Cache maintenance operations are required for different stages to 'see' the output of other stages in memory.
- *First access scope*: memory accesses by commands that *happen before* the barrier.
  - AKA: **srcAccessMask**
  - A barrier does a **cache clean (or flush)** on any caches used in the first access scope
- *Second access scope*: memory accesses by commands that *happen after* the barrier.
  - AKA: **dstAccessMask**
  - A barrier does a **cache invalidate** on any caches used in the first access scope

LUNARG

Types of synchronization errors

RAW	Read-after-write	This occurs when a subsequent operation uses the result of a previous operation without waiting for the result to be completed
WAR	Write-after-read	This occurs when a subsequent operation overwrites a memory location read by a previous operation before that operation is complete. (requires only execution dependency)
WAW	Write-after-write	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
WRW	Write-racing-write	This occurs when unsynchronized subpasses/queues perform writes to the same set of memory locations
RRW	Read-racing-write	This occurs when unsynchronized subpasses/queues perform read and write operations on the same set of memory locations

LUNARG

Hello race condition!

```
vkCmdCopyBuffer(cb, buffer_a, buffer_b, 1, &region); /* a is copy src */
vkCmdCopyBuffer(cb, buffer_c, buffer_a, 1, &region); /* a is copy dst */
```

Write after read hazard because buffer_a is both src and dst in commands with no execution dependency!

```
vkCmdCopyBuffer(cb, buffer_a, buffer_b, 1, &region);
vkCmdPipelineBarrier(cb, VK_PIPELINE_STAGE_TRANSFER_BIT,
                         VK_PIPELINE_STAGE_TRANSFER_BIT,
                         0, 0, nullptr, 0, nullptr, 0,nullptr);
vkCmdCopyBuffer(cb, buffer_c, buffer_a, 1, &region);
```


LUNARG

Pipeline Stages and Access Masks

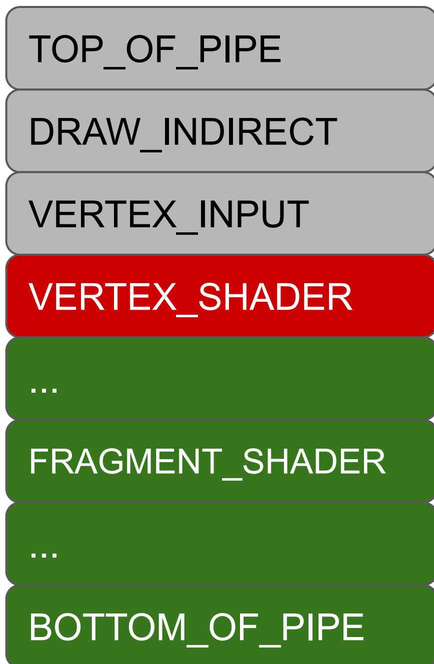
- Pipeline stages bits are ordered
  - Logical ordering defined in vulkan spec
  - In srcStageMask, each stage bit also waits for all earlier stages
  - In dstStageMask, each stage bit also blocks all later stages
  - You can often ‘get away’ with only setting some of the bits you are synchronizing
- Access mask bits are independent
  - You need to set ALL bits you are synchronizing
  - **BUT**, you must explicitly specify each pipeline stage if you want to use an access mask that requires it. (This is a common source of errors)

LUNARG

Pipeline Stage - details

- Was a 32-bit mask, but all bits are used so sync2 made it 64 bits
  - Several extensions **require** using sync2 or the 'special' stage mask bits
  - All bit values in the 32-bit mask have same meaning in the 64-bit mask
- Valid values are limited by
  - Queue Capabilities
  - Enabled extensions & features
  - Being in a renderpass
- Special values
  - NONE, TOP_OF_PIPE, BOTTOM_OF_PIPE - will be discussed separately
  - ALL_COMMANDS - blocks 'everything', all stages and some event-related commands
  - ALL_GRAPHICS - all active parts of the graphics pipeline
  - In sync2, several stages expand to multiple new stages
    - e.g. TRANSFER stage is equivalent to (COPY | BLIT | CLEAR | RESOLVE)

Pipeline stage ordering example



- In srcStageMask:
  - VERTEX_SHADER also waits on all grey stages
- In dstStageMask
  - VERTEX_SHADER also blocks all green stages

LUNARG

Pipeline barriers - execution dependency chains

```
Command A
Barrier1
Command B
Barrier2
Command C
// A, B, C should
// execute in order
```

- TOP_OF_PIPE
- DRAW_INDIRECT
- COMPUTE_SHADER
- BOTTOM_OF_PIPE

- Used to get sequential execution of multiple commands
- Barrier1.dstMask must the same as or after Barrier2.srcMask in pipeline stage order

Barrier1.dstMask	Barrier2.srcMask	dependency chain?
DRAW_INDIRECT	DRAW_INDIRECT	Yes
DRAW_INDIRECT	COMPUTE_SHADER	Yes
COMPUTE_SHADER	DRAW_INDIRECT	No
DRAW_INDIRECT	BOTTOM_OF_PIPE or ALL_COMMANDS	Yes (but might be slow)

LUNARG

Waiting for everything or nothing

- srcStageMask = ALL_COMMANDS blocks or waits for all stages
  - This is **wait for idle** on the GPU and will often hurt performance
- srcStageMask = NONE or TOP_OF_PIPE
  - **Your barrier waits for nothing**
  - Can only form an *execution dependency chain* with the prior barrier with dstStageMask = ALL_COMMANDS
- dstStageMask = NONE or BOTTOM_OF_PIPE
  - **Nothing can wait for your barrier**
  - Use srcStageMask = ALL_COMMANDS to form an *execution dependency chain*
- This comes up when interacting with other parts of Vulkan
  - Semaphores & Fences usually are OK
  - Renderpass implicit SubpassDependencies often go poorly (more later)

LUNARG

Access Mask details

- Was a 32-bit mask, but all bits are used so sync2 made it 64 bits
  - Several extensions **require** using sync2 or the 'special' access mask bits
  - All bit values in the 32-bit mask have same meaning in the 64 bit mask
- Valid bits are limited by which bits are set in the corresponding StageMask
  - Eg. PIPELINE_STAGE_TRANSFER allows ACCESS_TRANSFER_READ or WRITE
  - sync2 defines 200+ VUIDs to identify all possible errors
- Special values
  - NONE - no memory access, used to define an execution barrier
  - MEMORY_READ, MEMORY_WRITE - any memory access allowed by StageMask.
  - SHADER_READ- in sync2 expands to (SAMPLER_READ|STORAGE_READ|UNIFORM_READ)
  - SHADER_WRITE - in sync2 expands to STORAGE_WRITE (which is above 2^32)

LUNARG

Memory Barriers

```
typedef struct VkMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
} VkMemoryBarrier;

/* sync2 */
typedef struct VkMemoryBarrier2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkPipelineStageFlags2KHR    srcStageMask;
    VkAccessFlags2KHR           srcAccessMask;
    VkPipelineStageFlags2KHR    dstStageMask;
    VkAccessFlags2KHR           dstAccessMask;
} VkMemoryBarrier2KHR;
```

- A memory barrier synchronizes all memory accessible by the GPU
- You can use to synchronize buffers and images, UNLESS you are doing Image Layout Transition or Queue Family Ownership Transfer
- N barriers can be converted to 1 by or-ing all of their masks together
- Sync2 makes pipeline stages be part of the barrier structures instead of separate parameters to vkCmdPipelineBarrier()

LUNARG

Buffer Barriers - Queue Family Ownership

- Like a MemoryBarrier except
  - Adds a VkBuffer handle
  - Adds srcQueueFamilyIndex, dstQueueFamilyIndex for Queue Family Ownership (QFO) Transfer
- Queue Family Ownership Transfer (also part of Image Memory Barriers)
  - VK_SHARING_MODE_CONCURRENT is usually very slow (for Images only)
  - VK_SHARING_MODE_EXCLUSIVE requires a QFO barrier to switch ownership between one queue family and another
- QFO Barrier
  - Create a Buffer (or Image) Memory Barrier
  - Submit on src queue, only srcAccessMask used.
  - Submit on dst queue, only dstAccessMask used.
  - However, **both** PipelineStageMasks are used by both queues
  - Use a Semaphore to synchronize the 2 queues
  - Use the 'other' PipelineStageMask to form an execution dependency with the Semaphore wait or signal operation

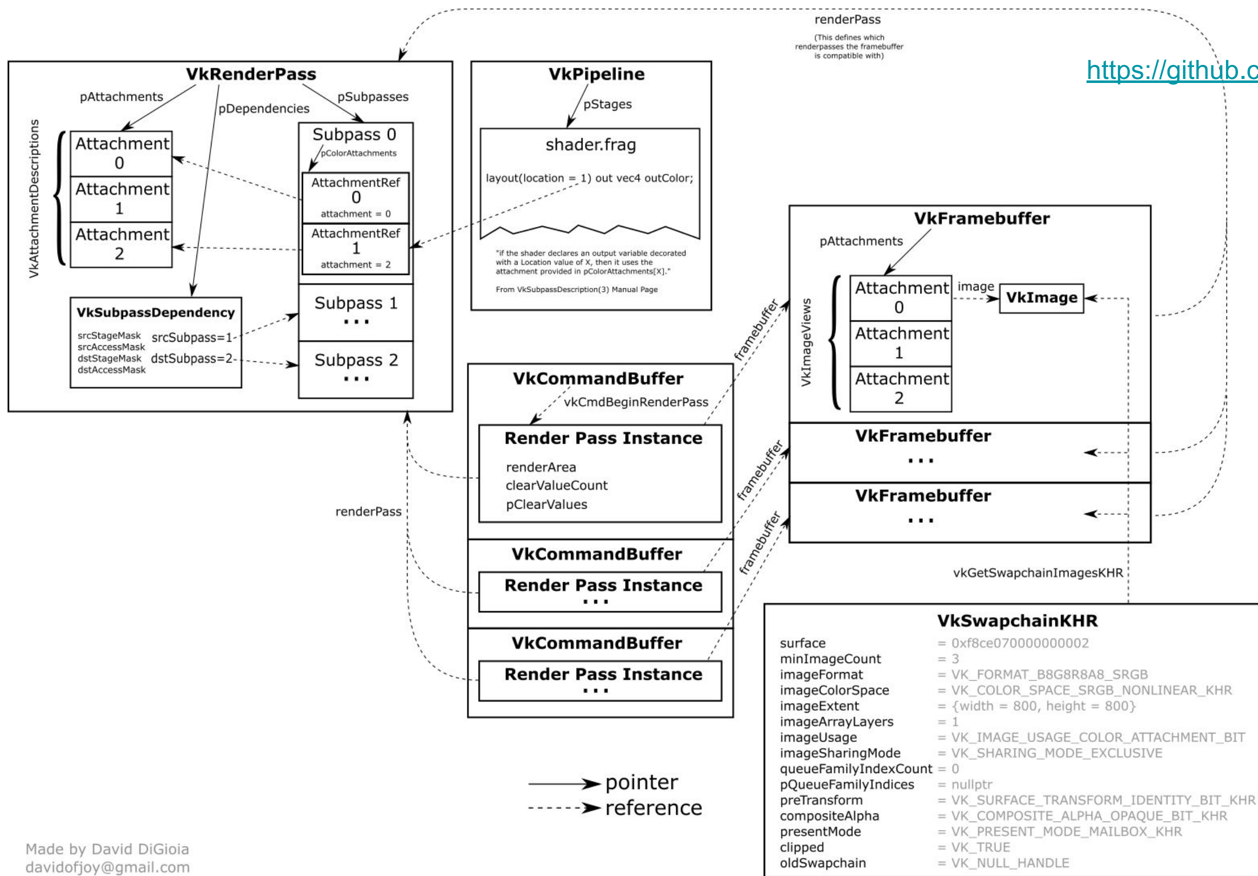
Image Memory Barriers

- Like a BufferMemoryBarrier except
  - VkImage handle instead of VkBuffer
  - Adds VkImageLayout oldLayout and newLayout to allow Image Layout Transitions
- Image Layout Transitions
  - Re-arrange memory for efficient use by different pipeline stages
  - *Happens between* 🤯 the first and second execution scopes of the barrier
  - Each subresource of an image can be transitioned independently.
- sync2 adds magic ‘do the right thing’ layouts
  - Avoid the need for providing different layouts for Color, Depth and Stencil Images
  - VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR
  - VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR

LUNARG

Renderpass - it is REALLY complicated

https://github.com/David-DiGioia/vulkan-diagrams



LUNARG

Renderpass - what to watch out for

- Load and store operations often cause synchronization errors
  - LOAD_OP_DONT_CARE generates WRITE accesses to your attachments
- Rasterization order synchronizes some operations within a subpass
- Pipeline Barriers in a RenderPass are even trickier
  - You need a Subpass self-Dependency (srcSubpass == dstSubpass) that includes all the pipeline stages your barrier(s) will use
  - Set of allowed pipeline stages is limited
- Internal Subpass Dependencies don't affect the outside world
  - (srcSubpass and dstSubpass != EXTERNAL)
  - *First and second execution scopes* only include commands in other subpasses
- Implicit External Subpass Dependencies don't do what you want, define them explicitly

LUNARG

Implicit Subpass External Dependencies

```
/* INITIAL implicit subpass */
VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
 // First subpass attachment is used in
    .dstSubpass = firstSubpass;
    .srcStageMask = NONE;
    .dstStageMask = ALL_COMMANDS;
    .srcAccessMask = 0;
    .dstAccessMask =
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dependencyFlags = 0;
};
/* FINAL implicit subpass is similar,
 * but src and dst are swapped
 */
```

- These are Image Memory Barriers for your attachments
- Inserted by the driver ONLY IF
  - You have initial or final layout transitions.
  - You don't provide your own dependency
- They wait for or block NOTHING
- Use them as a template but change the external side to be useful
  - You probably want an execution dependency chain!
- Maybe use different pipeline stages on internal side instead of ALL_COMMANDS
- Maybe **add** internal side access bits, but you probably want at least the default ones.

LUNARG

Other stuff

- Semaphores and Fences synchronize all GPU execution and memory accesses that were submitted before they signal
  - This is why wait for nothing / block nothing is sometimes ok
- Queries execute completely in submission order
- Many other changes in sync2:
  - vkQueueSubmit2KHR() is much nicer than vkQueueSubmit()
  - VkDependencyInfoKHR reduces typing for pipeline barrier commands
  - Event commands are redesigned

Q & A

Validating Vulkan Synchronization

John Zulauf, LunarG Inc.

Senior Graphics Software Engineer with 30 years Graphics experience across numerous platforms and from kernel drivers to application development.

Slides are available at:
https://www.lunarg.com/news-insights/white-papers/vulkan-synchronization-siggraph-2021/

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
- Some Limitations
  - Single Command Buffer (current release)
  - Limited aliasing detection (like kinds of resources)
  - No swizzle support
  - Not GPU Assisted (doesn’t know shader execution time information)
  - Limited extension support

LUNARG

Hazard Types (common)

- Read-after-write (RAW)
  - Operation uses the result of a previous operation without waiting for the result to be completed.
- Write-after-read (WAR)
  - Operation overwrites a memory location read by a previous operation before operation is complete.
- Write-after-write (WAW)
  - Operation writes to the same set of memory locations being written by a previous operation.

LUNARG

Hazard Types (Vulkan Specific)

- Write-racing-write (WRW)
  - Operations on unsynchronized subpasses/queues perform writes to the same set of memory locations.
- Read-racing-write (RRW)
  - Operations on unsynchronized subpasses/queues perform read and write operations on the same set of memory locations

Synchronization Validation Theory of Operation

- Tracks access history
  - At each byte
  - Operation Type (stage, access)
  - Stores “most recent” only (more below)
- Applies synchronization operations to access history
  - Identifies “safe” subsequent access operations
  - Track dependency chaining
- Validates accesses of each subsequent operation to access history
  - The stage and access for each are compared previous access and synchronization
  - Reports hazards
  - Any hazard reported earlier may mask detection of subsequent hazard with same memory

LUNARG

Using Synchronization Validation

- Clean Validation Run
    - Resolve all outstanding non-synchronization issues.
    - Recommend “best practices” and “GPU Assisted” as well.
- How To Enable
    - vkconfig
    - vk_layer_settings.txt
    - Environment variables
- Running
    - Disable all other validation
    - Chase down issues in debugger.
        - “Debug Action: Break” on Windows
        - Break in vkCreateDebugUtilsMessengerEXT callback

Simple Sync Val Demo

- Using the Vulkan-Samples

LUNARG

Congratulations, It’s An Error.

[SYNC-HAZARD-WRITE_AFTER_WRITE] Object 0: handle = 0x8483000000000025, type = VK_OBJECT_TYPE_IMAGE; | MessageID = 0xfdf9f5e1 | vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE for image barrier 0 VkImage 0x8483000000000025[]. Access info (usage: SYNC_IMAGE_LAYOUT_TRANSITION, prior_usage: SYNC_TRANSFER_TRANSFER_WRITE, write_barriers: 0, command: vkCmdCopyBufferToImage, seq_no: 2, reset_no: 1)

- Understanding the parts of this error will take a little background knowledge

Think Like Synchronization Validation

- Stage/Access pairs are need to describe the usages of resources
    - Not all pairs are valid, valid pairs expressed as enum SYNC_<STAGE>_<ACCESS>
    - Enum reflects Sync2 expanded pipeline stages
- How does the current operation (draw, transfer, etc.) affect the resource
    - Stage/access of operation for each resource
    - Comparison to earlier command stage/access and sync operations ("..is it safe?")
    - Include implicit operations (layout transition, load, resolve, store)
- 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
- What are the prior commands that touch a given resource (memory location)?
    - Include implicit operations (layout transition, load, resolve, store)

Synchronization Validation Messages

<command_name>: Hazard <hazard_type>
<command_specific_resource_identifier>
Access info (
    usage: <current_stage_access>,
    prior_usage: <most_recent_prior_stage_access>,
    (read_barriers|write_barriers): <cumulative_barrier_for_prior>,
    command: <command_of_prior_usage>,
    seq_no: <command_index_of_prior_command>,
    reset_no: <times_command_buffer_been_reset>
)

LUNARG

Congratulations, It's An Error. (annotated)

Current command.

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE
for image barrier 0 VkImage 0x8483000000000025[].
Access info (
usage: SYNC_IMAGE_LAYOUT_TRANSITION,
prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
write_barriers: 0,
command: vkCmdCopyBufferToImage,
seq_no: 2, reset_no: 1).

LUNARG

Congratulations, It's An Error. (annotated)

Current command.

Hazard Type

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE
for image barrier 0 VkImage 0x8483000000000025[].
Access info (
    usage: SYNC_IMAGE_LAYOUT_TRANSITION,
    prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
    write_barriers: 0,
    command: vkCmdCopyBufferToImage,
    seq_no: 2, reset_no: 1).

LUNARG

Congratulations, It's An Error. (annotated)

Current command.

Hazard Type

Command specific resource identifier

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE for image barrier 0 VkImage 0x8483000000000025[]. Access info (

usage: SYNC_IMAGE_LAYOUT_TRANSITION,
prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
write_barriers: 0,
command: vkCmdCopyBufferToImage,
seq_no: 2, reset_no: 1).

LUNARG

Congratulations, It's An Error. (annotated)

Current command.

Hazard Type

Command specific resource identifier

Current command Stage/Access

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE
for image barrier 0 VkImage 0x848300000000025[].
Access info (
    usage: SYNC_IMAGE_LAYOUT_TRANSITION,
    prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
    write_barriers: 0,
    command: vkCmdCopyBufferToImage,
    seq_no: 2, reset_no: 1).

Congratulations, It's An Error. (annotated)

Current command.

Hazard Type

Command specific resource identifier

Previous command Stage/Access

Current command Stage/Access

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE for image barrier 0 VkImage 0x8483000000000025[]. Access info (
usage: SYNC_IMAGE_LAYOUT_TRANSITION,
prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
write_barriers: 0,
command: vkCmdCopyBufferToImage,
seq_no: 2, reset_no: 1).

LUNARG

Congratulations, It’s An Error. (annotated)

Current command.

Hazard Type

Command specific resource identifier

Previous command Stage/Access

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE
for image barrier 0 VkImage 0x8483000000000025[].
Access info (
usage: SYNC_IMAGE_LAYOUT_TRANSITION,
prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
write_barriers: 0,
command: vkCmdCopyBufferToImage,
seq_no: 2, reset_no: 1).

Current command Stage/Access

Barriers applied since previous command

LUNARG

Congratulations, It’s An Error. (annotated)

Current command.

Hazard Type

Command specific resource identifier

Previous command Stage/Access

vkCmdPipelineBarrier: Hazard WRITE_AFTER_WRITE for image barrier 0 VkImage 0x8483000000000025[]. Access info (
usage: SYNC_IMAGE_LAYOUT_TRANSITION,
prior_usage: SYNC_TRANSFER_TRANSFER_WRITE,
write_barriers: 0,
command: vkCmdCopyBufferToImage,
seq_no: 2, reset_no: 1).

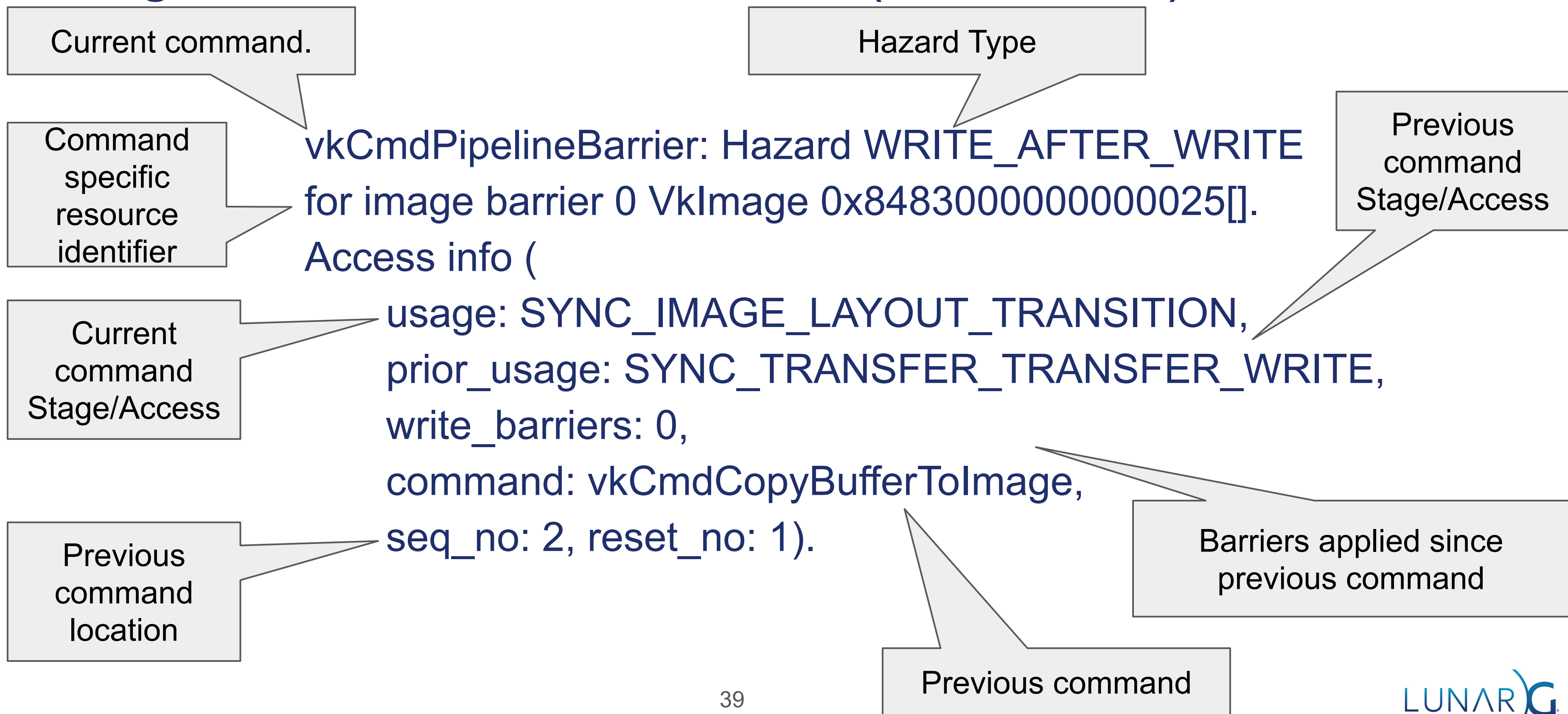
Current command Stage/Access

Barriers applied since previous command

Previous command

LUNARG

Congratulations, It's An Error. (annotated)



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

LUNARG

Simple Sync Val Demo Part II

- Using the Vulkan-Samples

LUNARG

Frequently Found Issues

- Missing pipeline stage for memory barriers
  - Stages are not logically extended for memory access barriers.
- Invalid stage/access pairs
  - Yields no barrier
- Relying on implicit subpass dependencies with VK_SUBPASS_EXTERNAL
  - Implicit Barriers are essential no-ops
- Missing memory dependencies with Image Layout Transitions
  - Transitions are full subresource range *read/write* operations.
- Missing stage/access scopes for load operations
  - Color and depth/stencil are done by different stage/access.

Debugging Strategies

- Stage/Access Completeness In Barriers
  - By inspection. Simplest approach.
  - Look at read/write barrier information vs. usage vs. existing barriers
- Localizing w/ Access info
  - prior_usage and (prior) command data can help identify access which hazard with current
  - (read|write)_barrier
- 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

LUNARG

Debugging Strategies (cont'd)

- 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
- Method of bisection
  - Insert "big hammer" Barriers/Subpass Dependency
    - Stage: VK_PIPELINE_STAGE_ALL_COMMANDS_BIT (VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT inside render pass)
    - 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
  - Be sure to remove after
- Be sure and check Core/Parameter Validation as you change code

After the presentation

Questions or presentation feedback?

Contact John Zulauf: @jzulauf on the Vulkan KhronosDevs slack channel

- https://app.slack.com/client/TDMDFS87M/CDTJ9BELF
- Or sign up for the KhronosDevs slack channel here:https://www.khronos.org/news/permalink/khronos-developer-slack-5bfc62eb261764.20435008

Report bugs or make feature requests here:
https://github.com/KhronosGroup/Vulkan-ValidationLayers

For more information:

- https://www.lunarg.com/news-insights/white-papers/guide-to-vulkan-synchronization-validation/
- https://www.lunarg.com/news-insights/white-papers/vulkan-synchronization2-validation/

Vulkan Synchronization -- SIGGRAPH 2021

Slides are available at:

https://www.lunarg.com/news-insights/white-papers/vulkan-synchronization-siggraph-2021/

LUNARG

Q & A

Backup

Hello race condition! (sync2)

```
vkCmdCopyBuffer(cb, buffer_a, buffer_b, 1, &region);

auto barrier = lvl_init_struct<VkMemoryBarrier2KHR>()
barrier.srcStageMask = VK_PIPELINE_STAGE_TRANSFER_BIT;
barrier.srcAccessMask = VK_ACCESS_NONE_KHR;
barrier.dstStageMask = VK_PIPELINE_STAGE_TRANSFER_BIT;
barrier.dstAccessMask = VK_ACCESS_NONE_KHR;

auto dep_info = lvl_init_struct<VkDependencyInfoKHR>();
dep_info.memoryBarrierCount = 1;
dep_info.pMemoryBarriers = &barrier;
vkCmdPipelineBarrier2KHR(cb, &dep_info);

vkCmdCopyBuffer(cb, buffer_c, buffer_a, 1, &region);
```

Events

- "Split" pipeline barriers
  - Can allow for more parallelism on the GPU
  - CmdSetEvent() is *first scope* (src)
  - CmdWaitEvents() is the *second scope* (dst)
- Hard to use (and infrequently used)
  - Only partially fixed by synchronization2
- Race conditions between Set, Reset, Wait commands
  - Require semaphore or pipeline barrier using the ALL_COMMANDS to avoid
  - "To fix correctly we need Timeline Events that work like Timeline Semaphores"
- May be signalled by Host
  - Sync2 VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR disables this

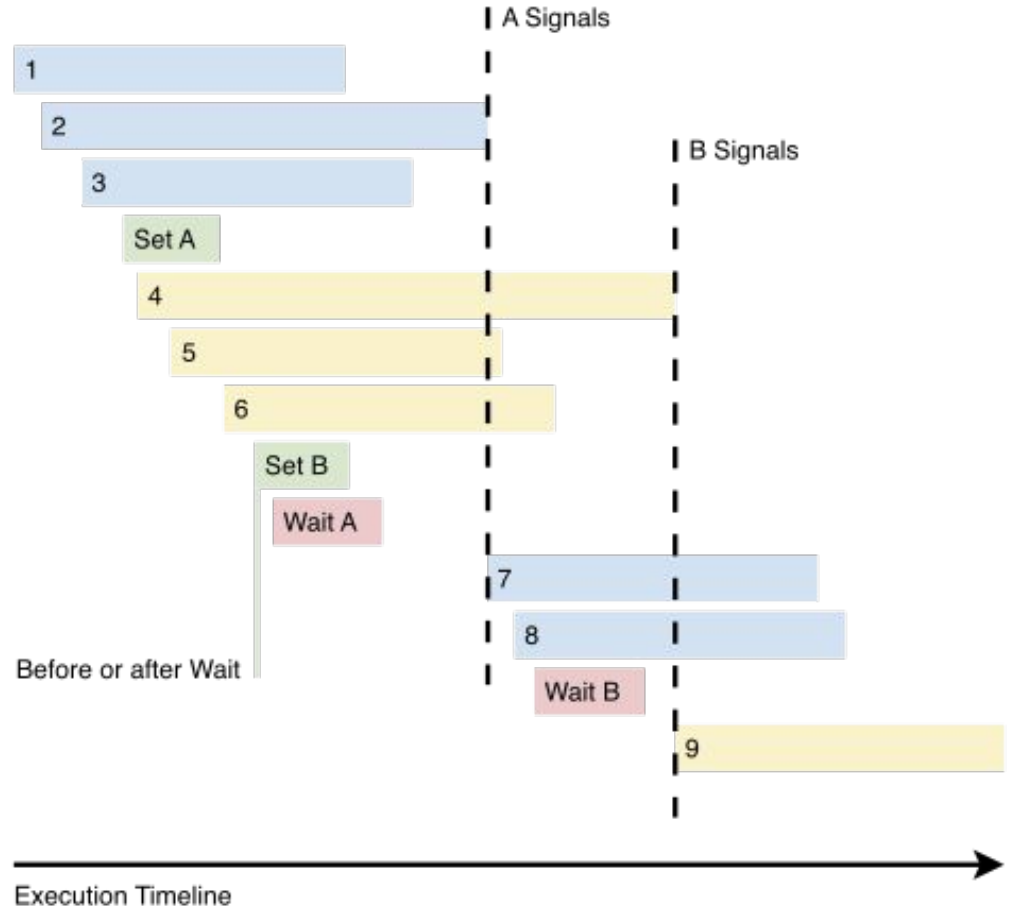
LUNARG

Events - example

Cmd 7 depends on 1, 2, & 3

Cmd 9 depends on 4, 5, & 6

Cmds 1-6 can potentially run in parallel which wouldn't be possible with pipeline barriers



LUNARG

Events - changes sync2

```
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent  event,
    VkPipelineStageFlags stageMask); /* src stage */

void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask); /* src stage */

void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags  srcStageMask,
    VkPipelineStageFlags dstStageMask,
    /* barrier lists omitted */);
```

- vkCmdSetEvent() only has enough information to set up execution dependencies.
- Driver cannot schedule work for memory dependencies until vkCmdWaitEvent() is called!

```
void vkCmdSetEvent2KHR(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    const VkDependencyInfoKHR* pDependencyInfo);

vkCmdResetEvent2KHR() same as vkCmdResetEvent()

void vkCmdWaitEvents2KHR(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    const VkDependencyInfoKHR* pDependencyInfos);
```

- vkCmdSetEvent2KHR() dependency info must match what is passed to vkCmdWaitEvent2KHR()
- each pDependencyInfo[i] has the barriers for pEvents[i]
    - In original function this was unclear

LUNARG