Guide to Vulkan Synchronization Validation

John Zulauf, LunarG
Version 1.0, released with SDK 1.2.162.1
January 2021

Table of Contents

Introduction

Synchronization Validation is implemented in the `VK_LAYER_KHRONOS_validation` layer as a Validation Object. When enabled, it is intended to identify resource access conflicts due to missing or incorrect synchronization operations between actions (Draw, Copy, Dispatch, Blit) reading or writing the same regions of memory.

Quick Start

The following quick start should enable initial testing for those familiar with Vulkan synchronization and debugging validation issues. Prior to enabling Synchronization Validation, assure that the default set of Validation checks run cleanly. Note: Quick Start readers should be familiar with both Vulkan Synchronization and using/configuring Vulkan Validation.

Running Synchronization Validation

The simplest way to run synchronization validation and debug issues is to:

- Enable Synchronization Validation using Vulkan Configurator (vkconfig).
- Create a debug callback with `vkCreateDebugUtilsMessengerEXT` with `VK_DEBUG_REPORT_ERROR_BIT_EXT` set.
- Set a breakpoint in the debug callback and run your application in the debugger.
- The hazards will be reported when a `vkCmd`... command with a hazard is recorded.

Synchronization Validation Messages

All synchronization error messages begin with `SYNC-<hazard name>`. The message body is constructed:

```
<cmd name>: Hazard <hazard name> <command specific details> Access info (<...>)
```

`Access info` contains information about current and prior usage (formatted `SYNC_<stage>_<access>`) and any intervening synchronization. Memory or subresource range of the usage is given in the command-specific details among other information.

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 operations, noting that color and depth/stencil are done by different stage/access.

Debugging Tips

- Read and write barriers in the error message can help identify the synchronization operation (either subpass dependency or pipeline barrier) with insufficient or incorrect destination stage/access masks (second scope).
- `Access info read_barrier` and `write_barrier` values of 0, reflect the absence of any barrier and can indicate an insufficient or incorrect source mask (first scope)
- Insert additional barriers with stage/access `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT`, `VK_ACCESS_MEMORY_READ_BIT|VK_ACCESS_MEMORY_WRITE_BIT` for both `src*Mask` and `dst*Mask` fields to locate missing barriers. If the inserted barrier *resolves* a hazard, the conflicting access *happens-before* the inserted barrier. (Be sure to delete later.)

What is synchronization and why is it important?

Correct synchronization is needed to ensure correct results from Vulkan operations (whether graphical or computational). Modern graphics hardware is both parallel and pipelined, with various operations happening simultaneously for performance reasons. Vulkan has a limited number of ordering guarantees, but for most operations, it is the application's responsibility to inform the implementation when ordering is required between operations. The need for such synchronization operations arises when the same region of memory is used by subsequent operations in different ways -- for

example a mip-level being written by a blit operation, and then being used for sampled lookup by a shader. If these two uses are not guaranteed to operate sequentially, a data hazard exists. These hazards are:

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

When an application is insufficiently synchronized, data corruption of various types can occur. Blit, copy, or present operations may result in a destination image (or buffer) that is based on an incomplete source image (or buffer), or one partially updated from a previous state. Unsynchronized buffers may contain uninitialized or invalid constant, vertex, or index values corrupting in results of a draw or dispatch call. As with all parallel processing problems, these corruptions may appear only rarely, or sporadically, making them difficult to find and debug.

Correct synchronization is also important for application portability. Vulkan implementations may vary in the *effective* synchronization implicit in their software or hardware. This means that applications operating without correct synchronization may work correctly versus a given manufacture, model, or even driver version, but may fail with data corruptions (again, potentially rarely, and sporadically) on other

implementations. Even an optimization introduced by a new driver could expose a failure mode, indicating missing or insufficient set of synchronization operations.

Synchronization Operations

Synchronization operations create dependencies between operations accessing memory. These execution and memory dependencies are used to solve data hazards, i.e. to ensure that read and write operations occur in a well-defined order. Write-after-read hazards can be solved with just an execution dependency, but read-after-write and write-after-write hazards need appropriate memory dependencies to be included between them. If an application does not include dependencies to solve these hazards, the results and execution orders of memory accesses are undefined.

While it is critical for data integrity that sufficient dependencies be defined to avoid these hazards, it is equally important that **excess** dependencies are not used, which would impact performance by overly serializing execution. Synchronization Validation can be used to test reductions in dependencies, with confidence that the less restrictive synchronization scheme is still correct.

Pointers to synchronization blogs/articles

Synchronization Examples

Keeping your GPU fed without getting bitten

Yet another blog explaining Vulkan synchronization

Understanding Synchronization Validation

When a Vulkan command that accesses memory is recorded, the accesses for that command are tested for potential conflicts with previous accesses -- the hazards listed above. These accesses may be explicitly defined by command parameters, for example copy and blit source and destination. The accesses may be defined by the creation parameters of another Vulkan object (for example a RenderPass, FrameBuffer, or ShaderModule), or by descriptor or buffer binding. In addition to copy, blit, draw, and dispatch calls which clearly imply memory accesses, synchronization operations (barriers, events) and render pass operations (begin, next, end) may have implicit

accesses. The user should not be surprised when hazards occur during (or due to) these other types of Vulkan commands.

When running Synchronization validation against your code, the current functionality will report hazards between accesses within the same command buffer. The error messages will list the command during which the hazard occurred, command specific details of which access within the command conflicts with a prior access, details about the current access, and the prior command and access with which the current access conflicts. The prior command is identified by command name and the sequence number within the current command buffer.

Stage/Access Usage Pairs

While the Vulkan specification typically lists pipeline stages and access types independently, only a small subset of the possible combinations of stage and access types are valid. (See *Table 4. Supported access types* in the Access Types section of the Vulkan API Specification). In error messages the stage/access pairs are given as `SYNC_<stage>_<access type>` for brevity, denoting an access of VK_ACCESS_<access_type>_BIT occurring on stage `VK_PIPELINE_STAGE_<stage>_BIT`. (Note that for extension bits, the extension tag is appended to the `SYNC_...` string.). Additional `SYNC_...` strings are defined for implicit accesses without stage or access. For example `SYNC_IMAGE_LAYOUT_TRANSITION` denotes the read/write access implied by an image layout transition.

Validation Checks

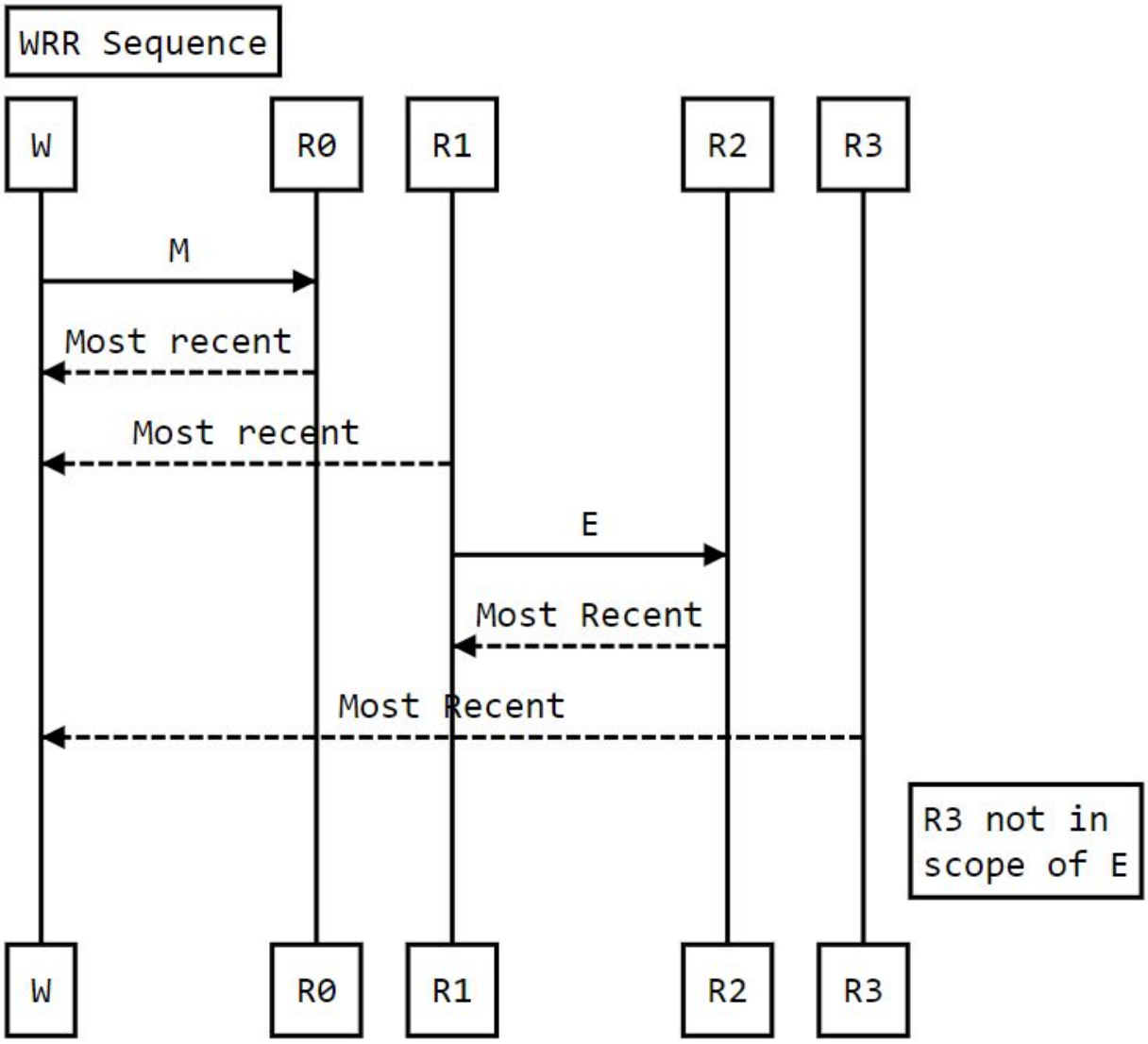
The Vulkan specification describes synchronization dependencies in terms of relationships between operations. Synchronization Validation however looks at the impact of synchronization operations on the safety (or correctness) of subsequent actions on resources, whether ranges of memory, or image subresource ranges and extents. For each Vulkan command that operates on memory, the prior state of affected memory ranges is inspected for the stage/access type of prior usage, the effect of synchronization operations on which subsequent usages are known safe relative to the prior accesses. Synchronization Validation error messages report hazards caused by the current Vulkan command's resource accesses relative to prior accesses for the same resources.

Most Recent Access

When reporting hazards, Synchronization Validation only inspects the *most recent* access for a given memory or image subresource range. It can be shown that for any hazard with an access *prior* to the most recent access, a hazard between the most recent access and that prior access must also exist. Since, all prior hazards are assumed to have been reported, tracking state prior to the most recent access is unneeded.

For read operations the most recent access rules apply to prior reads with execution barriers (or ordering) relative to the current read. The prior write access is only considered the most recent access if no intervening prior read has occurred that *happens-before* the current read. Consider the following sequence of access and barriers (listed in submission order) acting on the same memory address:

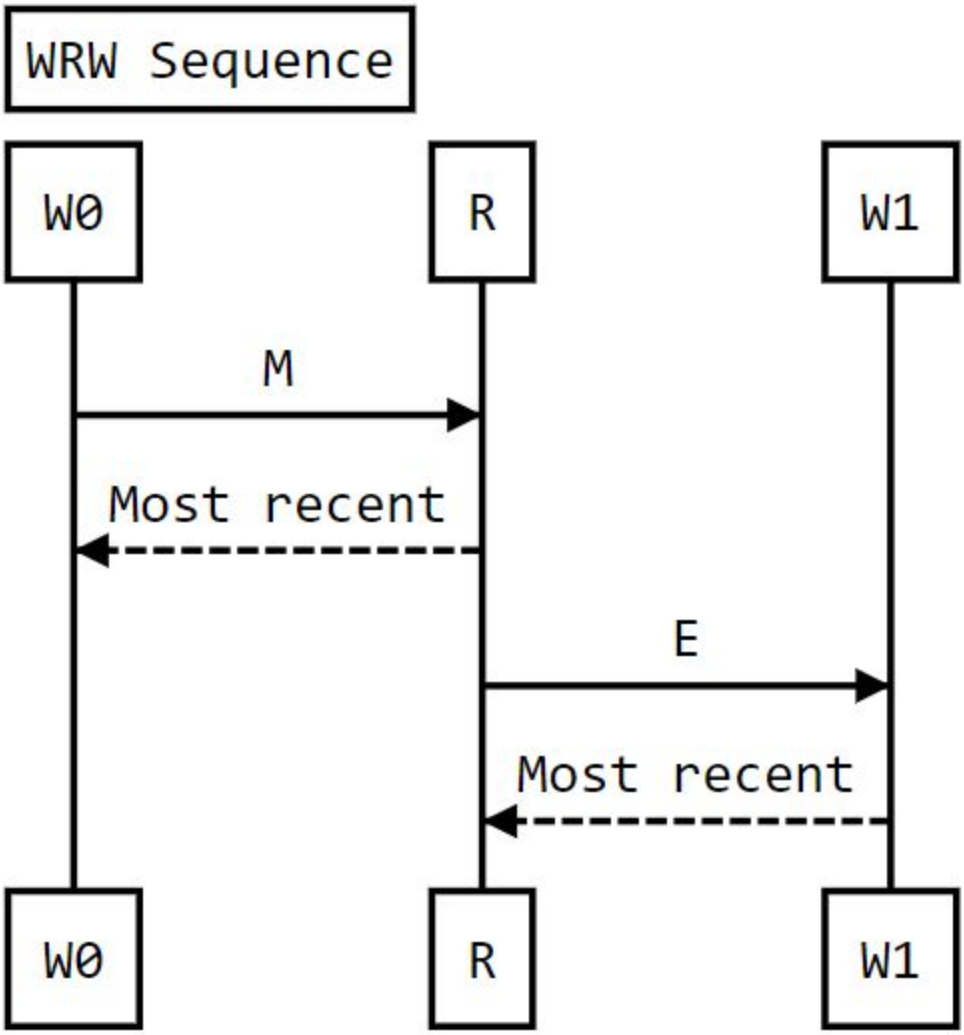
Operation	Description
W	write operation
M	memory barrier guarding access at R0
R0	first read operation
R1	second read operation
E	execution barrier such that R2 *happens-after R1*
R2	third read operation
R3	fourth read operation with stage not in second execution scope of E



For write hazard checks in a given range of memory addresses, if there are intervening read operations between the current write and the most recent previous write, these intervening read operations are considered the most recent access. In that case, write-after-write checks are not done.

Consider the following sequence of operations on the same memory address:

Operation	Description
W0	first write operation
M	memory barrier guarding access at R
R	read access
E	execution barrier guarding access at W1
W1	second write operation



In this case, a read-after-write check is done for R based on W0 and M, and a write-after-read check is performed on W1 based on R and E. W1 is not checked against W0 for write-after-write. If W0, M, R **is** not a hazard, this guarantees W0 is available and visible to R, and thus to any operations that *happen-after*. As such, the correctness of R, E, W1 depends solely on those operations. The correctness of the entire sequence can be assured by pairwise hazard checks.

Using Synchronization Validation

Before validating synchronization operations, resolve all validation errors from Standard Validation and Thread Safety. This will prevent wasted effort debugging hazards caused by invalid usage of the commands involved. It is possible to have entirely valid Vulkan command streams, and still have synchronization issues. However, invalid Vulkan commands may introduce synchronization issues by changing the effect of those commands.

Enabling synchronization validation.

The simplest way to enable Synchronization Validation is using Vulkan Configurator. In "Validation Settings" uncheck all "Validation Areas" except "Handle Wrapping" and "Synchronization." This can also be done using your layer settings file, `vk_layer_settings.txt`

```
khronos_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,VK_VALIDATION_FEATURE_DI
SABLE_THREAD_SAFETY_EXT
```

or through environment variables: (on non-Windows system replace "`;`" with "`:`")

```
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;VK_VAL
IDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT
```

The `VK_EXT_validation_features` extension can be used to enable Synchronization Validation programmatically at CreateInstance time, as shows in this code snippet:

```
VkValidationFeatureEnableEXT enables[] =
{VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT};
VkValidationFeatureDisableEXT disables[4] = {
        VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT,
        VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT,
        VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT,
        VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT
};
VkValidationFeaturesEXT features = {
```

```
        VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT, nullptr, 1, enables, 4, disables
};
VkInstanceCreateInfo info = {};
info.pNext = &features;
```

Typical Usage

Given the challenge of debugging synchronization issues, applications should be tested within a debugger, set to break for any validation error. On Windows this can be done in Vulkan Configurator by selecting “Debug/Action” options “Break” and “Debug Output” causes Synchronization Validation to show the text of the error message in the debugger and pause program execution for each error. On all platforms, this can be accomplished by setting a debug breakpoint in a debug callback. The debug callback is defined using `vkCreateDebugUtilsMessengerEXT` with `VK_DEBUG_REPORT_ERROR_BIT_EXT` set in `VkDebugReportCallbackCreateInfoEXT::flags`. For each reported hazard, the provoking Vulkan call will be on the debugger call stack, aiding debugging.

Understanding Synchronization Validation Messages

All synchronization error messages begin with `SYNC-<hazard name>` where <hazard name> is one of the hazard types listed above.

The message body for each is constructed:

`<cmd name>: Hazard <hazard name> <command specific details> Access info (<...>)`

Command specific details typically include the specifics of the access within the current command. Examples of typical command specific detail:

Command type	Details
Copy or blit	source or destination and region index
Draw or dispatch	Descriptor: binding, type Attachment: index and type Bound buffer: vertex or index
ImageBarriers	`oldLayout`, `newLayout`, subresource
Render pass	transitions `oldLayout`, `newLayout` load/store/resolve: attachment index, type, and operation

The `Access info` is common to all Synchronization Validation error messages. The fields in `Access info` are:

Field	Description
`usage`	The stage/access of the current command using the memory range
`prior_usage`	The stage/access of the previous (hazarded) memory use (same naming as `usage`)
`read_barrier`	For read `usage`, the list of stages with execution barriers between `prior_usage` and `usage`
`write_barrier`	For write `usage`, the list of stage/access (in `usage` format) with memory barriers between `prior_usage` and `usage`
`command`	The command that performed `prior_usage`
`seq_no`	The zero based index of `command` within the command buffer it is recorded to
`reset_no`	the reset count of the command buffer `command` is recorded to

Root Causing Hazards

While full details of the current resource access are available for the current Vulkan command, the information for locating `prior_usage` are more limited. Most applications (apart from debug and replay tools) will not have the sequence number (and traceback) of commands within the current command buffer, so the location of the access w.r.t. the `prior_usage` may take some effort to determine. Additionally, the `command` correlated with the `prior_usage` will likely be correct, and indicate that the synchronization operations between `command` and the current command are missing, incorrect, or incomplete.

An important note (repeated here from the quickstart), is that for *memory* barriers stage masks are *not* logically extended to logically earlier and later stages, but only apply to the specific stages specified. *Execution* barriers with a dstStageMask of `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT do imply an execution barrier including all later stages (for example VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT). However, assuming an access specified by dstAccessMask of VK_ACCESS_SHADER_READ_BIT would only define a memory barrier with the *vertex shader stage/shader read* stage/access. No *memory barrier* would be established with the *fragment shader stage* for the specified access.

For developer's intimate with the details of their rendering pipeline, this information may be sufficient to directly identify the cause of the hazard. However, the following approaches may help quickly refine the cause and fix for identified hazards.

From Access info information

The access information gives several useful pieces of information to help locate the missing synchronization operation. Using this information to find the synchronization can depend on the type of hazard and prior_usage.

Hazards vs. Prior Image Layout Transitions

If the prior_usage is SYNC_IMAGE_LAYOUT_TRANSITION, the write_barrier should help identify the vkCommandPipelineBarrier or VkSubpassDependency, of the prior_usage. The stages/access listed correlating to the dstStageMask and dstAccessMask for the prior command should match the write_barrier value. The fix is likely (barring missing dependency chain dstStageMask and chaining srcStageMask bits) to ensure that the current usage added to the dstStageMask and dstAccessMask fields of the barrier or renderpass operation that performed the layout transition.

Hazards at Image Layout Transitions

If the hazard occurs at a layout transition, you likely have no further to look than the `srcStageMask` and `srcAccessMask` of the current operation to find the missing or incorrect stage or access needed to guard the layout transition. Depending on code structure, the optimal solution may be to provide barriers at an earlier stage (when the presence or absence of a potential hazard would be known).

Hazards between buffer and/or image resource uses

When a resource changes between roles (being written or read) without an appropriate synchronization operation, a hazard will occur. Common situations are transitions between transfer and rendering (or compute) operations. If the `write_barrier` or `read_barrier` fields are non-zero, the actual error is likely a malformed barrier between usages. The `command` and `prior_usage` fields should help identify the previous access and the barrier field can be used to find the existing barrier, event, or renderpass operation similarly to finding prior image layout transitions. Barriers constructed from a chain of dependencies may be more difficult to back trace in this way and may require a different approach.

Method of bisection

In addition to back-tracking from `Access info` to find the missing or incomplete barriers, one can debug these issues by a simple bisection search. To determine where in program flow a missing barrier occurs simply insert a serializing debug barrier (described below) within your application *prior* to the hazard. If the hazard is removed, then the cause of the hazard is prior to the location of the debug barrier. If the hazard remains, the cause of the hazard is subsequent to the debug barrier.

A debug barrier is either a `vkCmdPipelineBarrier` or `VkSubpassDependency`, that specifies that all execution and access after the barrier *happen-after* all execution and access prior to it. In both cases set `srcStageMask` and `dstStageMask` set to `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT`. For the pipeline barrier one `VkMemoryBarrier` is required with both `srcAccessMask` and `dstAccessMask` set to `VK_ACCESS_MEMORY_READ_BIT|VK_ACCESS_MEMORY_WRITE_BIT`. The corresponding fields within `VkSubpassDependency` are set to the same value. If a `vkCmdPipelineBarrier` debug barrier is to be used during a subpass, it must all be specified as a self-dependency for that subpass. (See Subpass Self-dependency in the Vulkan API Specification)

Note: this is for debug only and should not be left in production code as this could have significant performance impact as the barrier serializes all GPU operations before and after the inserted barrier.

Optimizing Synchronization with Synchronization Validation

While synchronization is required to avoid data corruption within a Vulkan application, excessive synchronization can negatively impact performance by introducing stalls in command execution. Synchronization Validation can be used to identify the minimum needed barriers and dependencies. To do this, one can reduce the scopes of barriers and dependencies and note the hazards that are reported by Synchronization Validation. By adding just the barriers needed to eliminate the hazard error message, you can establish a minimally synchronized set of barriers and dependencies. Clearly one must be careful to exercise all possible Vulkan command sequences capable of producing different hazards, to assure that the narrowed set maintains correctness.

The minimally set of synchronization operations established may be quite broad (in terms of a large number of stages and accesses guarded). This can occur when the Vulkan usage pattern is highly variable before and after the synchronization operations. In these cases, it may be useful for the application to customize the barriers for the various use patterns. In addition, reordering processing steps to reduce the need for synchronization, may be of value.

In assessing the need to further optimize synchronization, performance benchmarks, and GPU usage tools should can and should be used. At each step however, testing with Synchronization Validation can assure that correctness is maintained.

For Further Information and Current Status

Please contact LunarG to let us know what you think of Synchronization Validation or to offer suggestions for future releases. The best way to reach us is via our GitHub repository using the link shown below:

https://github.com/KhronosGroup/Vulkan-ValidationLayers

Status, known limitations, and ongoing work on this project can be followed on the Synchronization Validation project page:

https://github.com/KhronosGroup/Vulkan-ValidationLayers/projects/5

Revision history

Revision Date	SDK Release	Comment
August 28, 2020	SDK 1.2.148.1	Alpha version, initial release date
January 11, 2021	SDK 1.2.162.1	V1.0 release. Supporting single command buffer hazard detection. Remove alpha specific comments, updated enable strings to standard versions, added project information