Maister's Graphics Adventures

Low-level graphics programming

Yet another blog explaining Vulkan synchronization

After playing Fire Emblem: Three Houses for an ungodly 160 hours over the past weeks, I guess it's time to put on my professor hat on the internet instead.

One topic I've been meaning to write about for a long time is synchronization in Vulkan. It's a large hurdle to overcome when learning the API, and rather than mechanically explaining how it works, my goal here is to instill a mental model in the reader. Despite its reputation for maddening complexity, it is actually understandable and quite logical once you get over the initial hurdles.

Where appropriate, I will use terms which match the Vulkan specification.

The Vulkan queue

For this part of the discussion we will only consider a single VkQueue. There is a lot to consider for single-queue synchronization, and dealing with multiple queues is a small extension on top of single-queue synchronization, which is covered at the end when discussing semaphores.

The Vulkan queue is simply an abstraction where command buffers are submitted and the GPU churns through commands. Let's get some common beginner mistakes out of the way first.

Command buffer misconceptions

Many developers seem to think that command buffer boundaries are somehow special in Vulkan. It is very important to clarify that for purposes of synchronization, everything submitted to a queue is simply a linear stream of commands. Any synchronization applies globally to a VkQueue, there is no concept of a only-inside-this-command-buffer synchronization.

Command overlap

The specification states that commands start execution in-order, but complete out-of-order. Don't get confused by this. The fact that commands start in-order is simply convenient language to make the spec language easier to write. **Unless you add synchronization yourself, all commands in a queue execute out of order. Reordering may happen across command buffers and even vkQueueSubmits.** This makes sense, considering that Vulkan only sees a linear stream of commands once you submit, it is a pitfall to assume that splitting command buffers or submits adds some magic synchronization for you.

NOTE: Unlike Vulkan, I do believe D3D12 disables any overlap across queue submits, but don't quote me on that. Might be something to consider if you're coming from D3D-land.

NOTE: Frame buffer operations inside a render pass happen in API-order, of course. This is a special exception which the spec calls out.

Pipeline stages

Every command you submit to Vulkan goes through a set of stages. These stages are represented in the VK_PIPELINE_STAGE enum. See chapter 6.1.2 in spec. When we synchronize work in Vulkan, we synchronize work happening in these pipeline stages as a whole, and not individual commands of work.

Draw calls, copy commands and compute dispatches all go through pipeline stages one by one.

The mysterious TOP_OF_PIPE and BOTTOM_OF_PIPE stages

A common stumbling block is the TOP_OF_PIPE and BOTTOM_OF_PIPE stages. These are essentially "helper" stages, which do no actual work, but serve some important purposes. Every command will first execute the TOP_OF_PIPE stage. This is basically the command processor on the GPU parsing the command. BOTTOM_OF_PIPE is where commands retire after all work has been done. TOP_OF_PIPE and BOTTOM_OF_PIPE are useful in specific scenarios, keep them in mind for later, as they are a little tricky and beginners make many mistakes with these.

In-queue execution barriers

Before we tackle memory barriers, we must fully understand execution barriers, as they are a subset of memory barriers. The primary mechanism in Vulkan to introduce execution barriers is the pipeline barrier.

To introduce the simplest form of an execution dependency we use a pipeline barrier:

```
void vkCmdPipelineBarrier(
    VkCommandBuffer                             commandBuffer,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    VkDependencyFlags                           dependencyFlags,
    uint32_t                                    memoryBarrierCount,
    const VkMemoryBarrier*                      pMemoryBarriers,
    uint32_t                                    bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*                pBufferMemoryBarriers,
    uint32_t                                    imageMemoryBarrierCount,
    const VkImageMemoryBarrier*                 pImageMemoryBarriers);
```

If we ignore the memory barriers and flags here, we're essentially left with two arguments, srcStageMask and dstStageMask. This represents the heart of the Vulkan synchronization model. We're splitting the command stream in two with a barrier, where we consider "everything before" the barrier, and "everything after" the barrier, and these two halves are synchronized in some way.

Section 6.1 lays this out in rather obtuse language, but we boil it down to:

srcStageMask

This represents what we are waiting for. Vulkan does not let you add fine-grained dependencies between individual commands. Instead you get to look at all work which happens in certain pipeline stages. For example, if we were to submit this series of commands starting off a fresh VkDevice:

- vkCmdDispatch (VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT)
- vkCmdCopyBuffer (VK_PIPELINE_STAGE_TRANSFER_BIT)
- vkCmdDispatch (VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT)

- vkCmdPipelineBarrier (srcStageMask = VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT)

We would be referring to the two vkCmdDispatch commands, as they perform their work in the COMPUTE stage. Even if we split these 4 commands into 4 different vkQueueSubmits, we would still consider the same commands for synchronization. Essentially, the work we are waiting for is **all commands which have ever been submitted to the queue including any previous commands in the command buffer we're recording.** srcStageMask then restricts the scope of what we are waiting for. Only work happening in COMPUTE_SHADER_BIT stage is relevant in this example. srcStageMask is a bit-mask as the name suggests, so it's perfectly fine to wait for both COMPUTE and TRANSFER work.

There are also flags to refer to "all commands", ALL_COMMANDS_BIT, which basically drains the entire queue for work. ALL_GRAPHICS_BIT is the same, but only for render passes.

NOTE: Here we will find a potential use case for TOP_OF_PIPE. srcStageMask of TOP_OF_PIPE is basically saying "wait for nothing", or to be more precise, we're waiting for the GPU to parse all commands, which is, a complete noop. We had to parse all commands before getting to the pipeline barrier command to begin with. When we get to memory barriers, this can be very useful.

dstStageMask

This represents the second half of the barrier. Any work submitted after this barrier will need to wait for the work represented by srcStageMask before it can execute. Only work in the specified stages are affected. For example, if dstStageMask is FRAGMENT_SHADER_BIT, vertex shading for future commands can begin executing early, we only need to wait once FRAGMENT_SHADER_BIT is reached.

NOTE: As an analog to srcStageMask with TOP_OF_PIPE, for dstStageMask, using BOTTOM_OF_PIPE can be kind of useful. This basically translates to "block the last stage of execution in the pipeline". Basically, we translate this to mean "no work after this barrier is going to wait for us". This might seem meaningless, but it will be useful when we discuss semaphores and memory barriers later.

A crude example

Let’s assume we record and submit some commands on a fresh VkDevice:

1. vkCmdDispatch
2. vkCmdDispatch
3. vkCmdDispatch
4. vkCmdPipelineBarrier(srcStageMask = COMPUTE, dstStageMask = COMPUTE)
5. vkCmdDispatch
6. vkCmdDispatch
7. vkCmdDispatch

With this barrier, the “before” set is commands {1, 2, 3}. The “after” set is {5, 6, 7}. A possible execution order here could be:

- #3
- #2
- #1
- #7
- #6
- #5

{1, 2, 3} can execute out-of-order, and so can {5, 6, 7}, but these two sets of commands can not interleave execution. In spec lingo {1, 2, 3} *happens-before* {5, 6, 7}.

https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples has some examples of how these stages are used in practice.

Events aka. split barriers

Vulkan provides a way to get overlapping work in-between barriers. The idea of VkEvent is to get some unrelated commands in-between the “before” and “after” set of commands, e.g.:

1. vkCmdDispatch
2. vkCmdDispatch
3. vkCmdSetEvent(event, srcStageMask = COMPUTE)
4. vkCmdDispatch
5. vkCmdWaitEvent(event, dstStageMask = COMPUTE)
6. vkCmdDispatch
7. vkCmdDispatch

The “before” set is now {1, 2}, and the after set is {6, 7}. 4 here is not affected by any synchronization and it can fill in the parallelism “bubble” we get when draining the GPU of work from 1, 2, 3. For advanced compute, this is a very important thing to know about, but not all GPUs and drivers can take advantage of this feature.

Execution dependency chain

This is a subtle – but very important – point which I don’t think is well enough understood. The general gist of it is that when we use dstStageMask to block stages, the dependencies in srcStageMask are carried forward into the blocked stages. Waiting for dstStageMask later will also wait for any dependencies dstStageMask had. It is easier to show an example here:

1. vkCmdDispatch
2. vkCmdDispatch
3. vkCmdPipelineBarrier(srcStageMask = COMPUTE, dstStageMask = TRANSFER)
4. vkCmdPipelineBarrier(srcStageMask = TRANSFER, dstStageMask = COMPUTE)
5. vkCmdDispatch
6. vkCmdDispatch

In this example we actually get a dependency between {1, 2} and {5, 6}. This is because we created a chain of dependencies between COMPUTE -> TRANSFER -> COMPUTE. When we wait for TRANSFER in 4. we must also wait for anything which is currently blocking TRANSFER. This might seem confusing, but it makes sense if we consider a slightly modified example.

1. vkCmdDispatch
2. vkCmdDispatch
3. vkCmdPipelineBarrier(srcStageMask = COMPUTE, dstStageMask = TRANSFER)
4. vkCmdMagicDummyTransferOperation
5. vkCmdPipelineBarrier(srcStageMask = TRANSFER, dstStageMask = COMPUTE)
6. vkCmdDispatch
7. vkCmdDispatch

In this scenario, it’s clear that {4} must wait for {1, 2}. And {6, 7} must wait for {4}. So, we have created a chain where {1, 2} -> {4} -> {6, 7}, and as {4} is noop, {1, 2} -> {6, 7} is achieved. That’s essentially the chain.

This has some uses when you want to "link up" barriers for whatever reason. I kinda wish Vulkan had some special "scoreboard" pipeline stages just for this use case …

Pipeline stages and render passes

COMPUTE and TRANSFER work is very simple when it comes to pipeline stages. The only stages they execute are:

- TOP_OF_PIPE
- DRAW_INDIRECT (for indirect compute only)
- COMPUTE / TRANSFER
- BOTTOM_OF_PIPE

Render passes are a bit more intricate, and it's very easy to confuse which pipelines stages do what.

In render passes there are two "families" of pipeline stages, those which concern themselves with geometry processing, and the fragment family, which does rasterization / frame buffer operations.

Aside from TOP_OF_PIPE/BOTTOM_OF_PIPE, we have

Geometry

- DRAW_INDIRECT – Parses indirect buffers
- VERTEX_INPUT – Consumes fixed function VBOs and IBOs
- VERTEX_SHADER – Actual vertex shader
- TESSELLATION_CONTROL_SHADER
- TESSELLATION_EVALUATION_SHADER
- GEOMETRY_SHADER

Fragment

- EARLY_FRAGMENT_TESTS
- FRAGMENT_SHADER
- LATE_FRAGMENT_TESTS
- COLOR_ATTACHMENT_OUTPUT

For the most part, it's the Fragment stages which are a bit confusing. Each of them have their own use cases.

EARLY_FRAGMENT_TESTS

This is the stage where early depth/stencil tests happen. This stage isn't all that useful or meaningful except in some very obscure scenarios with frame buffer self-dependencies (aka, GL_ARB_texture_barrier). This is also where a render pass performs its loadOp of a depth/stencil attachment.

LATE_FRAGMENT_TESTS

This is where late depth-stencil tests take place, and also where depth-stencil attachments are stored with storeOp when a render pass is done.

HELPFUL TIP ON FRAGMENT TEST STAGES

It's somewhat confusing to have two stages which basically do the same thing. When you're waiting for a depth map to have been rendered in an earlier render pass, you should use srcStageMask = LATE_FRAGMENT_TESTS_BIT, as that will wait for the storeOp to finish its work.

When blocking a render pass with dstStageMask, just use a mask of EARLY_FRAGMENT_TESTS | LATE_FRAGMENT_TESTS.

NOTE: dstStageMask = EARLY_FRAGMENT_TESTS alone might work since that will block loadOp, but there might be shenanigans with memory barriers if you are 100% pedantic about any memory access happening in LATE_FRAGMENT_TESTS. If you're blocking an early stage, it never hurts to block a later stage as well.

COLOR_ATTACHMENT_OUTPUT

This one is where loadOp, storeOp, MSAA resolves and frame buffer blend stage takes place, basically anything which touches a color attachment in a render pass in some way. If you're waiting for a render pass which uses color to be complete, use srcStageMask = COLOR_ATTACHMENT_OUTPUT, and similar for dstStageMask when blocking render passes from execution.

Memory barriers

Now that we have the basics for execution barriers, we can kick it up a notch and consider memory barriers.

Execution order and memory order are two different things. GPUs are notorious for having multiple, incoherent caches which all need to be carefully managed to avoid glitched out rendering. This means that just synchronizing execution alone is not enough to ensure that different units on the GPU can transfer data between themselves.

If you are familiar with how C++11 introduced memory order and atomics, it is a good start, but the C++11 memory model does not consider that memory access can be incoherent to my knowledge. All CPU memory is assumed to be coherent, but memory order is weak on basically anything non-x86. Vulkan expands on this concept.

The two concepts in the Vulkan specification we need to understand is memory being **available** and memory being **visible**. This is an abstraction over the fact that GPUs have incoherent caches. To explain this I will describe a mental model of a hypothetical GPU design which should make sense if you are familiar with how caches work.

NOTE: There is a formal Vulkan memory model now which covers all of this in extreme detail. I admit I have not studied it enough to make references to it here, but developers really don't need to know that level of detail to use Vulkan correctly.

The L2 cache / main memory

We will let the last cache hierarchy represent the "master" memory controller which all caches are connected to. This cache is connected to all other L1 caches, and external DDR memory. The GPU DDR memory is connected to the CPU memory controller in some way (PCI-e or UMA).

When our L2 cache contains the most up-to-date data there is, we can say that memory is **available**, because L1 caches connected to L2 can pull in the most up-to-date data there is.

Incoherent L1 caches

Vulkan specifies a bunch of flags in the VK_ACCESS_ series of enums. These flags represent memory access which can be performed. Each pipeline stage can perform certain memory access, and thus we take the combination of pipeline stage + access mask and we get potentially a very large number of incoherent caches on the system. Each GPU core has its own set of L1 caches as well.

Of course, real GPUs will only have a fraction of the possible caches here, but as long as we are explicit about this in the API, any GPU driver can simplify this as needed.

Under section 6.1.3, table 4 in the Vulkan spec you can see a list of all possible access masks which can be used with a pipeline stage. These access masks either read from a cache, or write to an L1 cache in our mental model.

We say that memory is **visible** to a particular stage + access combination if memory has been made **available** and we then make that memory **visible** to the relevant stage + access mask.

Once a shader stage writes to memory, the L2 cache no longer has the most up-to-date data there is, so that memory is no longer considered **available**. If other caches try to read from L2, it will see undefined data. Whatever wrote that data must make those writes **available** before the data can be made **visible** again.

Cache flush and invalidate

To be clear, we can say that "making memory available" is all about flushing caches, and "making memory visible" is invalidating caches. This should make it more obvious what is going on. I will use the spec terminology however.

VkMemoryBarrier

If we revisit vkCmdPipelineBarrier, we can pass in a list of global memory barriers.

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

A global memory barrier deals with access to any resource, and it's the simplest form of a memory barrier. This means that in vkCmdPipelineBarrier, we are specifying 4 things to happen in order:

- Wait for srcStageMask to complete
- Make all writes performed in possible combinations of srcStageMask + srcAccessMask **available**
- Make **available** memory **visible** to possible combinations of dstStageMask + dstAccessMask.
- Unblock work in dstStageMask.

A common misconception I see is that _READ flags are passed into srcAccessMask, but this is redundant. It does not make sense to make reads available, i.e. you don't flush caches when you're done reading data.

MEMORY ACCESS AND TOP_OF_PIPE/BOTTOM_OF_PIPE

Never use AccessMask != 0 with these stages. These stages **do not perform memory accesses**, so any srcAccessMask and dstAccessMask combination with either stage will be meaningless, and spec disallows this. TOP_OF_PIPE and BOTTOM_OF_PIPE are purely there for the sake of execution barriers, not memory barriers.

SPLIT MEMORY BARRIERS

A very important point here is that it's perfectly possible to split up the "make available" and "make visible" operations. This is similar to the execution dependency chain discussed earlier.

We can do something silly like:

- vkCmdDispatch – writes to an SSBO, VK_ACCESS_SHADER_WRITE_BIT
- vkCmdPipelineBarrier(srcStageMask = COMPUTE, dstStageMask = TRANSFER, srcAccessMask = SHADER_WRITE_BIT, dstAccessMask = 0)
- vkCmdPipelineBarrier(srcStageMask = TRANSFER, dstStageMask = COMPUTE, srcAccessMask = 0, dstAccessMask = SHADER_READ_BIT)
- vkCmdDispatch – read from the same SSBO, VK_ACCESS_SHADER_READ_BIT

While StageMask cannot be 0, AccessMask can be 0.

VkBufferMemoryBarrier

This is not very interesting, we're just restricting memory availability and visibility to a specific buffer. No GPU I know of actually cares, I think it makes more sense to just use VkMemoryBarrier rather than bothering with buffer barriers.

VkImageMemoryBarrier

Unlike VkBufferMemoryBarrier, this one is critical. You have to change image layouts at some point and this is done as part of an image memory barrier.

```
typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;
```

The interesting bits are oldLayout and newLayout. The layout transition happens in-between the **make available** and **make visible** stages of a memory barrier. The layout transition itself is considered a read/write operation, and the rules are basically that memory for the image must be **available** before the layout transition takes place. After a layout transition, that memory is automatically made **available** (but not **visible**!). Basically, think of the layout transition as some kind of in-place data munging which happens in L2 cache somehow.

A PRACTICAL TOP_OF_PIPE EXAMPLE

Now we can actually make a practical example with TOP_OF_PIPE. If we just allocated an image and want to start using it, what we want to do is to just perform a layout transition, but we don't need to wait for anything in order to do this transition. This is where TOP_OF_PIPE is useful. Let's say that we're allocating a fresh image, and we're going to use it in a compute shader as a storage image. The pipeline barrier looks like:

- srcStageMask = TOP_OF_PIPE – Wait for nothing
- dstStageMask = COMPUTE – Unblock compute after the layout transition is done
- srcAccessMask = 0 – This is key, there are no pending writes to flush out. This is the only way to use TOP_OF_PIPE in a memory barrier. It's important to note that freshly allocated memory in Vulkan is always considered available and visible to all stages

and access types. You cannot have stale caches when the memory was never accessed … What about recycled/aliased memory you ask? Excellent question, we'll cover that too later.
- oldLayout = UNDEFINED – Input is garbage
- newLayout = GENERAL – Storage image compatible layout
- dstAccessMask = SHADER_READ | SHADER_WRITE

A PRACTICAL BOTTOM_OF_PIPE EXAMPLE

My favourite example here is swapchain images. We have to transition them into VK_IMAGE_LAYOUT_PRESENT_SRC_KHR before passing the image over to the presentation engine.

After transitioning into this PRESENT layout, we're not going to touch the image again until we reacquire the image, so dstStageMask = BOTTOM_OF_PIPE is appropriate.

- srcStageMask = COLOR_ATTACHMENT_OUTPUT (assuming we rendered to swapchain in a render pass)
- srcAccessMask = COLOR_ATTACHMENT_WRITE
- oldLayout = COLOR_ATTACHMENT_OPTIMAL
- newLayout = PRESENT_SRC_KHR
- dstStageMask = BOTTOM_OF_PIPE
- dstAccessMask = 0

Having dstStageMask = BOTTOM_OF_PIPE and access mask being 0 is perfectly fine. We don't care about making this memory **visible** to any stage beyond this point. We will use semaphores to synchronize with the presentation engine anyways.

Implicit memory ordering – semaphores and fences

Semaphores and fences are quite similar things in Vulkan, but serve a different purpose. Semaphores facilitate GPU <-> GPU synchronization across Vulkan queues, and fences facilitate GPU -> CPU synchronization.

These objects are signaled as part of a vkQueueSubmit. However, one very important thing to note about semaphores and fences is how they interact with memory. To signal a semaphore or fence, all previously submitted commands to the queue must complete. If this were a regular pipeline barrier, we would have srcStageMask =

ALL_COMMANDS_BIT. However, **we also get a full memory barrier, in the sense that all pending writes are made available.** Essentially, srcAccessMask = MEMORY_WRITE_BIT.

Implicit memory guarantees on vkQueueSubmit

While signaling a fence or semaphore works like a full cache flush, submitting commands to the Vulkan queue, makes all memory access performed by host visible to all stages and access masks. Basically, submitting a batch issues a cache invalidation on host visible memory. A common mistake is to think that you need to do this invalidation manually when the CPU is writing into staging buffers or similar. Something like:

- srcStageMask = HOST
- srcAccessMask = HOST_WRITE_BIT
- dstStageMask = TRANSFER
- dstAccessMask = TRANSFER_READ

If the write happened before vkQueueSubmit, this is automatically done for you.

NOTE: This kind of barrier is necessary if you are using vkCmdWaitEvents where you wait for host to signal the event with vkSetEvent. In that case, you might be writing the necessary host data ***after*** *vkQueueSubmit was called, which means you need a pipeline barrier like this. This is not exactly a common use case, but it's important to understand when these API constructs are useful.*

Implicit memory guarantees when waiting for a semaphore

While signalling a semaphore makes all memory available, waiting for a semaphore makes memory visible. This basically means you do not need a memory barrier if you use synchronization with semaphores since signal/wait pairs of semaphores works like a full memory barrier. Let's see an example where queue 1 writes to an SSBO in compute, and consumes that buffer as a UBO in a fragment shader in queue 2. We're going to assume the buffer was created with QUEUE_FAMILY_CONCURRENT.

NOTE: If you need to transfer ownership to a different queue family, you need memory barriers, one in each queue to release/acquire ownership.

Queue 1

- vkCmdDispatch
- vkQueueSubmit(signal = my_semaphore)

There is no pipeline barrier needed here. Signalling the semaphore waits for all commands, and all writes in the dispatch are made available to the device before the semaphore is actually signaled.

Queue 2

- vkCmdBeginRenderPass
- vkCmdDraw
- vkCmdEndRenderPass
- vkQueueSubmit(wait = my_semaphore, pDstWaitStageMask = FRAGMENT_SHADER)

When we wait for the semaphore, we specify which stages should wait for this semaphore, in this case the FRAGMENT_SHADER stage. All relevant memory access is automatically made visible, so we can safely access UNIFORM_READ_BIT in FRAGMENT_SHADER stage, without having extra barriers. The semaphores take care of this automatically, nice!

Execution dependency chain with semaphore

Just like pipeline barriers having execution dependency chains, we can create execution dependency chains with semaphores as well. pDstWaitStageMask in vkQueueSubmit blocks certain stages from executing.

If we create a pipeline barrier with srcStageMask targeting one of the stages in the wait stage mask, we also wait for the semaphore to be signaled. This is extremely useful for doing image layout transitions on swapchain images. We need to wait for the image to be acquired, and only then can we perform a layout transition. The best way to do this is to use pDstWaitStageMask = COLOR_ATTACHMENT_OUTPUT_BIT, and then use srcStageMask = COLOR_ATTACHMENT_OUTPUT_BIT in a pipeline barrier which transitions the swapchain image after semaphore is signaled.

Host memory reads

While signalling a fence makes all memory available, it does not make them available to the CPU, just within the device. This is where dstStageMask = PIPELINE_STAGE_HOST and dstAccessMask = ACCESS_HOST_READ_BIT flags come in. If you intend to read back data to the CPU, you must issue a pipeline barrier which makes memory available to the HOST as well. In our mental model, we can think of this as flushing the GPU L2 cache out to GPU main memory, so that CPU can access it over some bus interface.

Safely recycling memory and aliasing memory

We earlier had an example with creating a fresh VkImage and transitioning it from UNDEFINED, and waiting for TOP_OF_PIPE. As explained, we did not need to specify any srcAccessMask since we knew that memory was guaranteed to be available. The reason for this is because of the implied guarantee of signalling a fence. In order to recycle memory, we must have observed that the GPU was done using the image with a fence. In order to signal that fence, any pending writes to that memory must have been made available, so even recycled memory can be safely reused without a memory barrier. This point is kind of subtle, but it really helps your sanity not having to inject memory barriers everywhere.

However, what if we consider we want to alias memory inside a command buffer? The rule here is that in order to safely alias, all memory access from the active alias must be made available before a new alias can take its place. Here's an example for a case where we have two VkImages which are used in two render passes, and they alias memory. When one image alias is written to, all other images immediately become "undefined". There are some exceptions in the specification for when multiple aliases can be valid at the same time, but for now we assume that is not the case.

- vkCmdPipelineBarrier(image = image1, oldLayout = UNDEFINED, newLayout = COLOR_ATTACHMENT_OPTIMAL, srcStageMask = COLOR_ATTACHMENT_OUTPUT, srcAccessMask = COLOR_ATTACHMENT_WRITE, dstStageMask = COLOR_ATTACHMENT_OUTPUT, dstAccessMask = COLOR_ATTACHMENT_WRITE|READ)

image1 will contain garbage here so we need to transition away from UNDEFINED. We need to make any pending writes to COLOR_ATTACHMENT_WRITE available before the layout transition takes place, assuming that we're running these commands every frame. The following render pass will wait for this transition to take place using dstStageMask/dstAccessMask.

- vkCmdBeginRenderPass/EndRenderPass
- vkCmdPipelineBarrier(image = image2, …)
- vkCmdBeginRenderPass/EndRenderPass

image1 was written to, so image2 was invalidated. Similar to the pipeline barrier for image1, we need to transition away from UNDEFINED. We need to make sure any write to image1 is made available before we can perform the transition. Next frame, image1 needs to take ownership again, and so on.

External subpass dependencies

Render passes in Vulkan have a concept of EXTERNAL subpass dependencies. This is arguably the most misunderstood aspect of Vulkan sync. I'd like to dedicate a section to this, because too many developers are lured into using it in cases where it's not terribly useful and very likely to cause bugs.

The main purpose of external subpass dependencies is to deal with initialLayout and finalLayout of an attachment reference. If initialLayout != layout used in the first subpass, the render pass is forced to perform a layout transition.

If you don't specify anything else, that layout transition will wait for nothing before it performs the transition. Or rather, the driver will inject a dummy subpass dependency for you with srcStageMask = TOP_OF_PIPE_BIT. This is not what you want since it's almost certainly going to be a race condition. You can set up a subpass dependency with the appropriate srcStageMask and srcAcessMask. **The external subpass dependency is basically just a vkCmdPipelineBarrier injected for you by the driver.** The whole premise here is that it's theoretically better to do it this way because the driver has more information, but this is questionable, at least on current hardware and drivers.

There is a very similar external subpass dependency setup for finalLayout. If finalLayout differs from the last use in a subpass, driver will transition into the final layout automatically. Here you get to change dstStageMask/dstAccessMask. If you do nothing here, you get BOTTOM_OF_PIPE/0, which can actually be just fine. A prime use case here is swapchain images which have finalLayout = PRESENT_SRC_KHR.

Essentially, you can ignore external subpass dependencies. Their added complexity give very little gain. Render pass compatibility rules also imply that if you change even minor things like which stages to wait for, you need to create new pipelines! This is dumb, and will hopefully be fixed at some point in the spec.

However, while the usefulness of external subpass dependencies is questionable, they have some convenient use cases I'd like to go over:

Automatically transitioning TRANSIENT_ATTACHMENT images

If you're on mobile, you should be using transient images where possible. When using these attachments in a render pass, it makes sense to always have them as initialLayout = UNDEFINED. Since we know that these images can only ever be used in COLOR_ATTACHMENT_OUTPUT or EARLY/LATE_FRAGMENT_TEST stages depending on their image format, the external subpass dependency writes itself, and we can just use transient attachments without having to think too hard about how to synchronize them. This is what I do in my Granite engine, and it's quite useful. Of course, we could just inject a pipeline barrier for this exact same purpose, but that's more boilerplate.

Automatically transitioning swapchain images

Typically, swapchain images are always just used once per frame, and we can deal with all synchronization using external subpass dependencies. We want initialLayout = UNDEFINED, and finalLayout = PRESENT_SRC_KHR.

srcStageMask is COLOR_ATTACHMENT_OUTPUT which lets us link up with the swapchain acquire semaphore. For this case, we will need an external subpass dependency. For the finalLayout transition after the render pass, we are fine with BOTTOM_OF_PIPE being used. We're going to use semaphores here anyways.

I also do this in Granite.

Conclusion

I hope this was useful. The post got a bit more mechanical than I hoped for, but it should be a more distilled version of the specification.

August 14, 2019 themaister Vulkan