KhronosGroup / **Vulkan-Docs**

Synchronization Examples

Tobin Ehlis edited this page on Sep 20 · 39 revisions

Synchronization in Vulkan can be confusing. It takes a lot of time to understand, and even then it's easy to trip up on small details. Most common use of Vulkan synchronization can be boiled down to a handful of use cases though, and this page lists a number of examples. May expand this in future to include other questions about synchronization.

Note that examples are usually expressed as a pipeline barrier, but events or subpass dependencies can be used similarly.

Compute to Compute Dependencies

First dispatch writes to a storage buffer, second dispatch reads from that storage buffer.

```
vkCmdDispatch(...);

VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT };

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

vkCmdDispatch(...);
```

First dispatch reads from a storage buffer, second dispatch writes to that storage buffer.

WAR hazards don't need a memory barrier between them - execution barriers are sufficient. A pipeline barrier or event without a memory barrier is an execution-only dependency.

```
vkCmdDispatch(...);

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    ...);
```

```
vkCmdDispatch(...);
```

First dispatch writes to a storage image, second dispatch reads from that storage image.

```
vkCmdDispatch(...);

// Storage image to storage image dependencies are always in GENERAL layout; no need for a layout transition
VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

vkCmdDispatch(...);
```

Three dispatches. First dispatch writes to a storage buffer, second dispatch writes to non-overlapping region of same storage buffer, third dispatch reads both regions.

```
vkCmdDispatch(...);
vkCmdDispatch(...);

VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT };

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

vkCmdDispatch(...);
```

Three dispatches. First dispatch writes to one storage buffer, second dispatch writes to a different storage buffer, third dispatch reads both.

Identical to previous example - global memory barrier covers all resources. Generally considered more efficient to do a global memory barrier than per-resource barriers, per-resource barriers should usually be used for queue ownership transfers and image layout transitions - otherwise use global barriers.

```
vkCmdDispatch(...);
vkCmdDispatch(...);

VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT };

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

vkCmdDispatch(...);
```

Compute to Graphics Dependencies

Note that interactions with graphics should ideally be performed by using subpass dependencies (external or otherwise) rather than pipeline barriers, but most of the following examples are still described as pipeline barriers for brevity.

Dispatch writes into a storage buffer. Draw consumes that buffer as an index buffer.

```
vkCmdDispatch(...);

VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_INDEX_READ_BIT };

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,   // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

... // Render pass setup etc.

vkCmdDraw(...);
```

Dispatch writes into a storage buffer. Draw consumes that buffer as an index buffer. A further compute shader reads from the buffer as a uniform buffer.

```
vkCmdDispatch(...);
```

```
// Batch barriers where possible if it doesn't change how synchronization takes place
VkMemoryBarrier memoryBarrier1 = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_INDEX_READ_BIT | VK_ACCESS_UNIFORM_READ_BIT};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT |
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier1,                      // pMemoryBarriers
    ...);

... // Render pass setup etc.

vkCmdDraw(...);

... // Render pass teardown etc.

vkCmdDispatch(...);
```

Dispatch writes into a storage buffer. Draw consumes that buffer as a draw indirect buffer.

```
vkCmdDispatch(...);

VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT };

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, // srcStageMask
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT,  // dstStageMask
    1,                                    // memoryBarrierCount
    &memoryBarrier,                       // pMemoryBarriers
    ...);

... // Render pass setup etc.

vkCmdDrawIndirect(...);
```

Dispatch writes into a storage image. Draw samples that image in a fragment shader.

```
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_GENERAL,
  .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,  // srcStageMask
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, // dstStageMask
    ...
    1,                                     // imageMemoryBarrierCount
    &imageMemoryBarrier,                   // pImageMemoryBarriers
    ...);


... // Render pass setup etc.

vkCmdDraw(...);
```

Dispatch writes into a storage texel buffer. Draw consumes that buffer as a draw indirect buffer, and then again as a uniform buffer in the fragment shader.

```
  vkCmdDispatch(...);

  VkMemoryBarrier memoryBarrier = {
    ...
    .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
    .dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT | VK_ACCESS_UNIFORM_READ_BIT};

  vkCmdPipelineBarrier(
      ...
      VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,    // srcStageMask
      VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT |
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, // dstStageMask
      1,                                       // memoryBarrierCount
      &memoryBarrier,                          // pMemoryBarriers
      ...);

  ... // Render pass setup etc.

  vkCmdDrawIndirect(...);
```

Graphics to Compute Dependencies

Draw writes to a color attachment. Dispatch samples from that image.

Note that color attachment write is NOT in the fragment shader, it has its own dedicated pipeline stage!

```
  vkCmdDraw(...);
```

```
... // Render pass teardown etc.

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,          // dstStageMask
    ...
    1,                                             // imageMemoryBarrierCount
    &imageMemoryBarrier,                           // pImageMemoryBarriers
    ...);

vkCmdDispatch(...);
```

Draw writes to a depth attachment. Dispatch samples from that image.

Note that depth attachment write is NOT in the fragment shader, it has its own dedicated pipeline stages!

```
vkCmdDraw(...);

... // Render pass teardown etc.

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT,   // srcStageMask
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,        // dstStageMask
    ...
    1,                                           // imageMemoryBarrierCount
    &imageMemoryBarrier,                         // pImageMemoryBarriers
    ...);

vkCmdDispatch(...);
```

Graphics to Graphics Dependencies

Many graphics to graphics dependencies can be expressed as a subpass dependency within a render pass, which is usually more efficient than a pipeline barrier or event. Where this is possible in the below, the example is expressed in terms of a subpass dependency.

First draw writes to a depth attachment. Second draw reads from it as an input attachment in the fragment shader.

The transition from VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL to VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL happens automatically as part of executing the render pass.

```
// Set this to the index in VkRenderPassCreateInfo::pAttachments where the depth image is described.
uint32_t depthAttachmentIndex = ...;

VkSubpassDescription subpasses[2];

VkAttachmentReference depthAttachment = {
    .attachment = depthAttachmentIndex,
    .layout     = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL};

// Subpass containing first draw
subpasses[0] = {
    ...
    .pDepthStencilAttachment = &depthAttachment,
    ...};

VkAttachmentReference depthAsInputAttachment = {
    .attachment = depthAttachmentIndex,
    .layout     = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL};

// Subpass containing second draw
subpasses[1] = {
    ...
    .inputAttachmentCount = 1,
    .pInputAttachments = &depthAsInputAttachment,
    ...};

VkSubpassDependency dependency = {
    .srcSubpass = 0,
    .dstSubpass = 1,
    .srcStageMask = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
                    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT,
    .dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
    .srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT,
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT,
    .dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT};

// If initialLayout does not match the layout of the attachment reference in the first subpass, there will be an implicit transition before starting the render pass.
// If finalLayout does not match the layout of the attachment reference in the last subpass, there will be an implicit transition at the end.
VkAttachmentDescription depthFramebufferAttachment = {
    ...
    .initialLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
    .finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL};

VkRenderPassCreateInfo renderPassCreateInfo = {
    ...
    .attachmentCount = 1,
    .pAttachments = &depthFramebufferAttachment,
    .subpassCount = 2,
    .pSubpasses = subpasses,
    .dependencyCount = 1,
    .pDependencies = &dependency};

vkCreateRenderPass(...);

...
```

First draw writes to a depth attachment. Second draw samples from that depth image in the fragment shader (e.g. shadow map rendering).

```
vkCmdDraw(...);

... // First render pass teardown etc.

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT,   // srcStageMask
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,       // dstStageMask
    ...
    1,                                           // imageMemoryBarrierCount
    &imageMemoryBarrier,                         // pImageMemoryBarriers
    ...);

... // Second render pass setup etc.

vkCmdDraw(...);
```

First draw writes to a color attachment. Second draw reads from it as an input attachment in the fragment shader.

```
// Set this to the index in VkRenderPassCreateInfo::pAttachments where the color image is described.
uint32_t colorAttachmentIndex = ...;

VkSubpassDescription subpasses[2];

VkAttachmentReference colorAttachment = {
    .attachment = colorAttachmentIndex,
    .layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};

// Subpass containing first draw
```

```
subpasses[0] = {
    ...
    .colorAttachmentCount = 1,
    .pColorAttachments = &colorAttachment,
    ...};

VkAttachmentReference colorAsInputAttachment = {
    .attachment = colorAttachmentIndex,
    .layout     = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL};

// Subpass containing second draw
subpasses[1] = {
    ...
    .inputAttachmentCount = 1,
    .pInputAttachments = &colorAsInputAttachment,
    ...};

VkSubpassDependency dependency = {
    .srcSubpass = 0,
    .dstSubpass = 1,
    .srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
    .dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
    .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT,
    .dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT};

// If initialLayout does not match the layout of the attachment reference in the first subpass, there will be an implicit transition before starting the render pass.
// If finalLayout does not match the layout of the attachment reference in the last subpass, there will be an implicit transition at the end.
VkAttachmentDescription colorFramebufferAttachment = {
    ...
    .initialLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
    .finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL};

VkRenderPassCreateInfo renderPassCreateInfo = {
    ...
    .attachmentCount = 1,
    .pAttachments = &colorFramebufferAttachment,
    .subpassCount = 2,
    .pSubpasses = subpasses,
    .dependencyCount = 1,
    .pDependencies = &dependency};

vkCreateRenderPass(...);

...
```

First draw writes to a color attachment. Second draw samples from that color image in the fragment shader.

```
vkCmdDraw(...);

... // First render pass teardown etc.

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // srcStageMask
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,         // dstStageMask
    ...
    1,                                             // imageMemoryBarrierCount
    &imageMemoryBarrier,                           // pImageMemoryBarriers
    ...);

... // Second render pass setup etc.

vkCmdDraw(...);
```

First draw writes to a color attachment. Second draw samples from that color image in the vertex shader.

```
  vkCmdDraw(...);

  ... // First render pass teardown etc.

  VkImageMemoryBarrier imageMemoryBarrier = {
    ...
    .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
    .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
    .oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
    .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
    /* .image and .subresourceRange should identify image subresource accessed */};

  vkCmdPipelineBarrier(
      ...
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // srcStageMask
      VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,         // dstStageMask
      ...
      1,                                             // imageMemoryBarrierCount
      &imageMemoryBarrier,                           // pImageMemoryBarriers
      ...);

  ... // Second render pass setup etc.

  vkCmdDraw(...);
```

First draw samples a texture in the fragment shader. Second draw writes to that texture as a color attachment.

This is a WAR hazard, which you would usually only need an execution dependency for - meaning you wouldn't need to supply any memory barriers. In this case you still need a memory barrier to do a layout transition though, but you don't need any access types in the src access mask. The layout transition itself is considered a write operation though, so you do need the destination access mask to be correct - or there would be a WAW hazard between the layout transition and the color attachment write.

```
vkCmdDraw(...);

... // First render pass teardown etc.

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = 0,
  .dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,         // srcStageMask
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // dstStageMask
    ...
    1,                                             // imageMemoryBarrierCount
    &imageMemoryBarrier,                           // pImageMemoryBarriers
    ...);

... // Second render pass setup etc.

vkCmdDraw(...);
```

Transfer Dependencies

Upload data from the CPU to a vertex buffer

Discrete Host and Device Memory

If there is a memory type with "HOST_VISIBLE" and not "DEVICE_LOCAL", and a separate type with "DEVICE_LOCAL" on, then use the following setup path. UMA systems are described in the next code block, though this code will work on such systems at the cost of additional memory overhead.

Setup:

```
// Data and size of that data
const uint32_t vertexDataSize = ... ;
const void* pData = ... ;

// Create a staging buffer for upload
VkBufferCreateInfo stagingCreateInfo = {
    ...
    .size = vertexDataSize,
    .usage = VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    ... };

VkBuffer stagingBuffer;
vkCreateBuffer(device, &stagingCreateInfo, NULL, &stagingBuffer);

// Create the vertex buffer
VkBufferCreateInfo vertexCreateInfo = {
    ...
    .size = vertexDataSize,
    .usage = VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
    ... };

VkBuffer vertexBuffer;
vkCreateBuffer(device, &vertexCreateInfo, NULL, &vertexBuffer);

...

// Allocate and memory bind memory for these buffers.
// Ensure that the staging buffer uses a memory type that has
// VK_MEMORY_PROPERTY_HOST_VISIBLE property and doesn't have
// VK_MEMORY_PROPERTY_DEVICE_LOCAL.
// The vertex buffer memory should be the opposite - it should include
// VK_MEMORY_PROPERTY_DEVICE_LOCAL and should not have
// VK_MEMORY_PROPERTY_HOST_VISIBLE.
// Use the example code documented in the description of
// VkPhysicalDeviceMemoryProperties:
// https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkPhysicalDeviceMemoryProperties.html

...

// Map the staging buffers - if you plan to re-use these (which you should),
// keep them mapped.
// Ideally just map the whole range at once as well.

void* stagingData;

vkMapMemory(
    ...
    stagingMemory,
    stagingMemoryOffset,
    vertexDataSize,
    0,
    &stagingData);

// Write data directly into the mapped pointer
fread(stagingData, vertexDataSize, 1, vertexFile);

// Flush the memory range
// If the memory type of stagingMemory includes VK_MEMORY_PROPERTY_HOST_COHERENT, skip this step

// Align to the VkPhysicalDeviceProperties::nonCoherentAtomSize
uint32_t alignedSize = (vertexDataSize-1) - ((vertexDataSize-1) % nonCoherentAtomSize) + nonCoherentAtomSize;
```

```
// Setup the range
VkMappedMemoryRange stagingRange = {
    ...
    .memory = stagingMemory,
    .offset = stagingMemoryOffset,
    .size   = alignedSize};

// Flush the range
vkFlushMappedMemoryRanges(device, 1, &stagingRange);
```

Command Buffer Recording and Submission for a unified transfer/graphics queue:

```
vkBeginCommandBuffer(...);

// Submission guarantees the host write being complete, as per
// https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#synchronization-submission-host-writes
// So no need for a barrier before the transfer

// Copy the staging buffer contents to the vertex buffer
VkBufferCopy vertexCopyRegion = {
    .srcOffset = stagingMemoryOffset,
    .dstOffset = vertexMemoryOffset,
    .size      = vertexDataSize};

vkCmdCopyBuffer(
    commandBuffer,
    stagingBuffer,
    vertexBuffer,
    1,
    &vertexCopyRegion);


// If the graphics queue and transfer queue are the same queue
if (isUnifiedGraphicsAndTransferQueue)
{
    // If there is a semaphore signal + wait between this being submitted and
    // the vertex buffer being used, then skip this pipeline barrier.

    // Pipeline barrier before using the vertex data
    // Note that this can apply to all buffers uploaded in the same way, so
    // ideally batch all copies before this.
    VkMemoryBarrier memoryBarrier = {
        ...
        .srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT,
        .dstAccessMask = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT};

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TRANSFER_BIT ,      // srcStageMask
        VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,   // dstStageMask
        1,                                    // memoryBarrierCount
        &memoryBarrier,                       // pMemoryBarriers
        ...);


    vkEndCommandBuffer(...);

    vkQueueSubmit(unifiedQueue, ...);
}
else
{
    // Pipeline barrier to start a queue ownership transfer after the copy
    VkBufferMemoryBarrier bufferMemoryBarrier = {
        ...
        .srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT,
        .dstAccessMask = 0,
        .srcQueueFamilyIndex = transferQueueFamilyIndex,
        .dstQueueFamilyIndex = graphicsQueueFamilyIndex,
        .buffer = vertexBuffer,
        ...};

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TRANSFER_BIT ,      // srcStageMask
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, // dstStageMask
        1,                                    // bufferMemoryBarrierCount
        &bufferMemoryBarrier,                 // pBufferMemoryBarriers
        ...);


    vkEndCommandBuffer(...);

    // Ensure a semaphore is signalled here which will be waited on by the graphics queue.
    vkQueueSubmit(transferQueue, ...);

    // Record a command buffer for the graphics queue.
    vkBeginCommandBuffer(...);

    // Pipeline barrier before using the vertex buffer, after finalising the ownership transfer
    VkBufferMemoryBarrier bufferMemoryBarrier = {
        ...
        .srcAccessMask = 0,
        .dstAccessMask = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT,
        .srcQueueFamilyIndex = transferQueueFamilyIndex,
        .dstQueueFamilyIndex = graphicsQueueFamilyIndex,
        .buffer = vertexBuffer,
        ...};

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,    // srcStageMask
        VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,   // dstStageMask
        ...
        1,                                    // bufferMemoryBarrierCount
        &bufferMemoryBarrier,                 // pBufferMemoryBarriers
        ...);
    vkEndCommandBuffer(...);

    vkQueueSubmit(graphicsQueue, ...);
}
```

Unified Memory

For UMA systems, you can use the above, but it will use less memory if you avoid the staging buffer for these systems, as per the following setup. There is no need to perform any device-side synchronization assuming the first commands that use it are submitted *after* the upload (rather than using VkEvents, which are not recommended, and not described here).

Setup:

```
// Data and size of that data
const uint32_t vertexDataSize = ... ;
const void* pData = ... ;

// Create the vertex buffer
VkBufferCreateInfo vertexCreateInfo = {
    ...
    .size = vertexDataSize,
    .usage = VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
    ... };

VkBuffer vertexBuffer;
vkCreateBuffer(device, &vertexCreateInfo, NULL, &vertexBuffer);

...

// Allocate and memory bind memory for this buffer.
// It should use a memory type that includes HOST_VISIBLE, and ideally also
// DEVICE_LOCAL if available.
// Use the example code documented in the description of
// VkPhysicalDeviceMemoryProperties:
// https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkPhysicalDeviceMemoryProperties.html

...

// Map the vertex buffer

void* vertexData;

vkMapMemory(
    ...
    vertexMemory,
    vertexMemoryOffset,
    vertexDataSize,
    0,
    &vertexData);

// Write data directly into the mapped pointer
```

```
fread(vertexData, vertexDataSize, 1, vertexFile);

// Flush the memory range
// If the memory type of vertexMemory includes VK_MEMORY_PROPERTY_HOST_COHERENT, skip this step

// Align to the VkPhysicalDeviceProperties::nonCoherentAtomSize
uint32_t alignedSize = (vertexDataSize-1) - ((vertexDataSize-1) % nonCoherentAtomSize) + nonCoherentAtomSize;

// Setup the range
VkMappedMemoryRange vertexRange = {
    ...
    .memory = vertexMemory,
    .offset = vertexMemoryOffset,
    .size   = alignedSize};

// Flush the range
vkFlushMappedMemoryRanges(device, 1, &vertexRange);

// You may want to skip this if you're going to modify the
// data again
vkUnmapMemory(device, vertexMemory);
```

Upload data from the CPU to an image sampled in a fragment shader

This path is universal to both UMA and discrete systems, as images should be converted to optimal tiling on upload.

Setup:

```
// Data and size of that data
const uint32_t imageDataSize = ... ;

// Create a staging buffer for upload
VkBufferCreateInfo stagingCreateInfo = {
    ...
    .size = imageDataSize,
    .usage = VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    ... };

VkBuffer stagingBuffer;
vkCreateBuffer(device, &stagingCreateInfo, NULL, &stagingBuffer);

// Create the sampled image
VkImageCreateInfo imageCreateInfo = {
    ...
    // Set the dimensions for the image as appropriate
    .tiling = VK_IMAGE_TILING_OPTIMAL,
    .usage  = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
    ... };

VkImage image;
vkCreateImage(device, &imageCreateInfo, NULL, &image);

...
```

```
// Allocate and memory bind memory for these resources.
// Ensure that the staging buffer uses a memory type that has
// VK_MEMORY_PROPERTY_HOST_VISIBLE property and doesn't have
// VK_MEMORY_PROPERTY_DEVICE_LOCAL.
// The image memory should be the opposite - it should include
// VK_MEMORY_PROPERTY_DEVICE_LOCAL and should not have
// VK_MEMORY_PROPERTY_HOST_VISIBLE.
// Use the example code documented in the description of
// VkPhysicalDeviceMemoryProperties:
// https://www.khronos.org/registry/vulkan/specs/1.0/man/html/VkPhysicalDeviceMemoryProperties.html

...

// Map the staging buffers - if you plan to re-use these (which you should),
// keep them mapped.
// Ideally just map the whole range at once as well.

void* stagingData;

vkMapMemory(
    ...
    stagingMemory,
    stagingMemoryOffset,
    imageDataSize,
    0,
    &stagingData);

// Write data directly into the mapped pointer
fread(stagingData, imageDataSize, 1, imageFile);

// Flush the memory range
// If the memory type of stagingMemory includes VK_MEMORY_PROPERTY_HOST_COHERENT, skip this step

// Align to the VkPhysicalDeviceProperties::nonCoherentAtomSize
uint32_t alignedSize = (imageDataSize-1) - ((imageDataSize-1) % nonCoherentAtomSize) + nonCoherentAtomSize;

// Setup the range
VkMappedMemoryRange stagingRange = {
    ...
    .memory = stagingMemory,
    .offset = stagingMemoryOffset,
    .size   = alignedSize};

// Flush the range
vkFlushMappedMemoryRanges(device, 1, &stagingRange);
```

Command Buffer Recording and Submission:

```
vkBeginCommandBuffer(...);

// Submission guarantees the host write being complete, as per
// https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#synchronization-submission-host-writes
[illegible]
```

```
// So no need for a barrier before the transfer for that purpose, but one is
// required for the image layout changes.

// Pipeline barrier before the copy to perform a layout transition
VkImageMemoryBarrier preCopyMemoryBarrier = {
    ...
    .srcAccessMask = 0,
    .dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT,
    .oldLayout = VK_IMAGE_LAYOUT_UNDEFINED,
    .newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    .srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
    .dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
    .image = image,
    .subresourceRange = ... }; // Transition as much of the image as you can at once.

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, // srcStageMask
    VK_PIPELINE_STAGE_TRANSFER_BIT,    // dstStageMask
    ...
    1,                                 // imageMemoryBarrierCount
    &preCopyMemoryBarrier,             // pImageMemoryBarriers
    ...);


// Setup copies for the all regions required (should be batched into a single call where possible)
vkCmdCopyBufferToImage(
    commandBuffer,
    stagingBuffer,
    image,
    ... };

// If the graphics queue and transfer queue are the same queue
if (isUnifiedGraphicsAndTransferQueue)
{
    // Pipeline barrier before using the vertex data
    VkImageMemoryBarrier postCopyMemoryBarrier = {
        ...
        .srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT,
        .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
        .oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
        .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        .srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .image = image,
        .subresourceRange = ... }; // Transition as much of the image as you can at once.

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TRANSFER_BIT ,       // srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, // dstStageMask
        ...
        1,                                     // imageMemoryBarrierCount
        &postCopyMemoryBarrier,                // pImageMemoryBarriers
        ...);
    vkEndCommandBuffer(...);

    vkQueueSubmit(unifiedQueue, ...);
}
else
{
    // Pipeline barrier before using the vertex data
    VkImageMemoryBarrier postCopyMemoryBarrier = {
        ...
        .srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT,
        .dstAccessMask = 0,
        .oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
        .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        .srcQueueFamilyIndex = transferQueueFamilyIndex,
        .dstQueueFamilyIndex = graphicsQueueFamilyIndex,
        .image = image,
        .subresourceRange = ... }; // Transition as much of the image as you can at once.

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TRANSFER_BIT ,       // srcStageMask
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,  // dstStageMask
        ...
        1,                                     // imageMemoryBarrierCount
        &postCopyMemoryBarrier,                // pImageMemoryBarriers
        ...);

    vkEndCommandBuffer(...);

    vkQueueSubmit(transferQueue, ...);

    vkBeginCommandBuffer(...);

    // Pipeline barrier before using the vertex data
    VkImageMemoryBarrier postCopyMemoryBarrier = {
        ...
        .srcAccessMask = 0,
        .dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
        .oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
        .newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
        .srcQueueFamilyIndex = transferQueueFamilyIndex,
        .dstQueueFamilyIndex = graphicsQueueFamilyIndex,
        .image = image,
        .subresourceRange = ... }; // Transition as much of the image as you can at once.

    vkCmdPipelineBarrier(
        ...
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,     // srcStageMask
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, // dstStageMask
        ...
        1,                                     // imageMemoryBarrierCount
        &postCopyMemoryBarrier,                // pImageMemoryBarriers
        ...);
    vkEndCommandBuffer(...);

    vkQueueSubmit(graphicsQueue, ...);
}
```

Interactions with semaphores

If you have a dependency where the two commands being synchronized have a semaphore signal/wait between them, the additional synchronization done by pipeline barriers/events/subpass dependencies can be reduced or removed. Only parameters affected by the presence of the semaphore dependency are listed

Any dependency where only buffers are affected, or images where the layout doesn't change

```
// Nothing to see here - semaphore alone is sufficient.
// No additional synchronization required - remove those barriers.
```

Signalling a semaphore waits for all stages to complete, and all memory accesses are made available automatically. Similarly, waiting for a semaphore will make all memory accesses available, and prevent further work from being started until it is signalled. Note that in the case of QueueSubmit there is an explicit set of stages to prevent running in VkSubmitInfo::pWaitDstStageMask - for all other semaphore uses execution of all work is prevented.

Dependency between images where a layout transition is required, expressed before the semaphore signal

```
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .dstAccessMask = 0};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, // dstStageMask
    ...);

... // Semaphore signal/wait happens here

vkCmdDispatch(...);
```

Dependency between images where a layout transition is required, expressed after the semaphore signal

This example assumes that whatever stages are in dstStageMask are included in the VkSubmitInfo::pWaitDstStageMask defined for the relevant semaphore wait operation - otherwise this modification does not apply.

```
vkCmdDispatch(...);

... // Semaphore signal/wait happens here
```

```
VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = 0};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, // srcStageMask
    ...);


vkCmdDispatch(...);
```

Swapchain Image Acquire and Present

Combined Graphics/Present Queue

```
VkAttachmentReference attachmentReference = {
    .attachment = 0,
    .layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};

// Subpass containing first draw
VkSubpassDescription subpass = {
    ...
    .colorAttachmentCount = 1,
    .pColorAttachments = &attachmentReference,
    ...};

/* Only need a dependency coming in to ensure that the first
   layout transition happens at the right time.
   Second external dependency is implied by having a different
   finalLayout and subpass layout. */
VkSubpassDependency dependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL,
    .dstSubpass = 0,
    // .srcStageMask needs to be a part of pWaitDstStageMask in the WSI semaphore.
    .srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
    .dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
    .srcAccessMask = 0,
    .dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
    .dependencyFlags = 0};

/* Normally, we would need an external dependency at the end as well since we are changing layout in finalLayout,
   but since we are signalling a semaphore, we can rely on Vulkan's default behavior,
   which injects an external dependency here with
   dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
   dstAccessMask = 0. */

VkAttachmentDescription attachmentDescription = {
    ...
    .loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    .storeOp = VK_ATTACHMENT_STORE_OP_STORE,
    ...
    // The image will automatically be transitioned from UNDEFINED to COLOR_ATTACHMENT_OPTIMAL for rendering, then out to PRESENT_SRC_KHR at the end.
    .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED,
    // Presenting images in Vulkan requires a special layout.
    .finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR};

VkRenderPassCreateInfo renderPassCreateInfo = {
    ...
    .attachmentCount = 1,
    .pAttachments    = &attachmentDescription,
    .subpassCount    = 1,
    .pSubpasses      = &subpass,
    .dependencyCount = 1,
    .pDependencies   = &dependency};

vkCreateRenderPass(...);

...

vkAcquireNextImageKHR(
    ...
    acquireSemaphore,   //semaphore
    ...
    &imageIndex);       //image index

VkPipelineStageFlags waitDstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;

VkSubmitInfo submitInfo = {
    ...
    .waitSemaphoreCount = 1,
    .pWaitSemaphores = &acquireSemaphore,
    .pWaitDstStageMask = &waitDstStageMask,
    ...
    .signalSemaphoreCount = 1,
    .pSignalSemaphores = &graphicsSemaphore};

vkQueueSubmit(..., &submitInfo, ...);

VkPresentInfoKHR presentInfo = {
    .waitSemaphoreCount = 1,
    .pWaitSemaphores = &graphicsSemaphore,
    ...};

vkQueuePresentKHR(..., &presentInfo);
```

Multiple Queues

If the present queue is a different queue from the queue where rendering is done, a queue ownership transfer must additionally be performed between the two queues at both acquire and present time, which requires additional synchronization.

Render pass setup:

```
VkAttachmentReference attachmentReference = {
    .attachment = 0,
    .layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};

// Subpass containing first draw
VkSubpassDescription subpass = {
    ...
    .colorAttachmentCount = 1,
    .pColorAttachments = &attachmentReference,
    ...};

VkAttachmentDescription attachmentDescription = {
    ...
    .loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE,
    .storeOp = VK_ATTACHMENT_STORE_OP_STORE,
    ...
    .initialLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
    .finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};

/*  Due to these necessary extra synchronization points, it makes more sense
    to omit the sub pass external dependencies (which can't express a queue
    transfer), and batch the relevant operations with the new pipeline
    barriers we're introducing. */

VkRenderPassCreateInfo renderPassCreateInfo = {
    ...
    .attachmentCount = 1,
    .pAttachments    = &attachmentDescription,
    .subpassCount    = 1,
    .pSubpasses      = &subpass,
    .dependencyCount = 0,
    .pDependencies   = NULL};

vkCreateRenderPass(...);
```

Rendering command buffer - graphics queue

```
/* Queue ownership transfer is only required when we need the content to remain valid across queues.
   Since we are transitioning from UNDEFINED -- and therefore discarding the image contents to begin with --
   we are not required to perform an ownership transfer from the presentation queue to graphics.

   This transition could also be made as an EXTERNAL -> subpass #0 render pass dependency as shown earlier. */

VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = 0,
  .dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
  .oldLayout = VK_IMAGE_LAYOUT_UNDEFINED,
  .newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  .srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
  .dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
  /* .image and .subresourceRange should identify image subresource accessed */};
```

```
vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // srcStageMask
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // dstStageMask
    ...
    1,                                             // imageMemoryBarrierCount
    &imageMemoryBarrier,                           // pImageMemoryBarriers
    ...);


... // Render pass submission.

// Queue release operation. dstAccessMask should always be 0.
VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
  .dstAccessMask = 0,
  .oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,
  .srcQueueFamilyIndex = graphicsQueueFamilyIndex, // index of the graphics queue family
  .dstQueueFamilyIndex = presentQueueFamilyIndex,  // index of the present queue family
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT, // srcStageMask
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,          // dstStageMask
    ...
    1,                                             // imageMemoryBarrierCount
    &imageMemoryBarrier,                           // pImageMemoryBarriers
    ...);
```

Pre-present commands - presentation queue

```
// After submitting the render pass...
VkImageMemoryBarrier imageMemoryBarrier = {
  ...
  .srcAccessMask = 0,
  .dstAccessMask = 0,
  // A layout transition which happens as part of an ownership transfer needs to be specified twice one for the release, and one for the acquire.
  // No srcAccessMask is needed, waiting for a semaphore does that automatically.
  // No dstAccessMask is needed, signalling a semaphore does that automatically.
  .oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  .newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,
  .srcQueueFamilyIndex = graphicsQueueFamilyIndex, // index of the graphics queue family
  .dstQueueFamilyIndex = presentQueueFamilyIndex,  // index of the present queue family
  /* .image and .subresourceRange should identify image subresource accessed */};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,    // srcStageMask
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, // dstStageMask
    ...
    1,                                  // imageMemoryBarrierCount
    &imageMemoryBarrier,                // pImageMemoryBarriers
    ...);
```

Queue submission:

```
vkAcquireNextImageKHR(
    ...
    acquireSemaphore,   //semaphore
    ...
    &imageIndex);       //image index

VkPipelineStageFlags waitDstStageMask1 = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
VkSubmitInfo submitInfo1 = {
    ...
    .waitSemaphoreCount = 1,
    .pWaitSemaphores = &acquireSemaphore,
    .pWaitDstStageMask = &waitDstStageMask1,
    .commandBufferCount = 1,
    .pCommandBuffers = &renderingCommandBuffer,
    .signalSemaphoreCount = 1,
    .pSignalSemaphores = &graphicsSemaphore};

vkQueueSubmit(renderQueue, &submitInfo1, ...);

VkPipelineStageFlags waitDstStageMask2 = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
VkSubmitInfo submitInfo2 = {
    ...
    .waitSemaphoreCount = 1,
    .pWaitSemaphores = &graphicsSemaphore,
    .pWaitDstStageMask = &waitDstStageMask2,
    .commandBufferCount = 1,
    .pCommandBuffers = &prePresentCommandBuffer,
    .signalSemaphoreCount = 1,
    .pSignalSemaphores = &ownershipPresentSemaphore};

vkQueueSubmit(presentQueue, &submitInfo2, ...);

VkPresentInfoKHR presentInfo = {
    .waitSemaphoreCount = 1,
    .pWaitSemaphores = &ownershipPresentSemaphore,
    ...};

vkQueuePresentKHR(..., &presentInfo);
```

Full pipeline barrier

You should **ONLY USE THIS FOR DEBUGGING** - this is not something that should ever ship in real code, this will flush and invalidate all caches and stall everything, it is a tool not to be used lightly!

That said, it can be *really* handy if you think you have a race condition in your app and you just want to serialize everything so you can debug it.

Note that this does not take care of image layouts - if you're debugging you can set the layout of all your images to GENERAL to overcome this, but again - do not do this in release code!

```
VkMemoryBarrier memoryBarrier = {
  ...
  .srcAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT |
                   VK_ACCESS_INDEX_READ_BIT |
                   VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT |
                   VK_ACCESS_UNIFORM_READ_BIT |
                   VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                   VK_ACCESS_SHADER_READ_BIT |
                   VK_ACCESS_SHADER_WRITE_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 |
                   VK_ACCESS_TRANSFER_READ_BIT |
                   VK_ACCESS_TRANSFER_WRITE_BIT |
                   VK_ACCESS_HOST_READ_BIT |
                   VK_ACCESS_HOST_WRITE_BIT,
  .dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT |
                   VK_ACCESS_INDEX_READ_BIT |
                   VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT |
                   VK_ACCESS_UNIFORM_READ_BIT |
                   VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                   VK_ACCESS_SHADER_READ_BIT |
                   VK_ACCESS_SHADER_WRITE_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 |
                   VK_ACCESS_TRANSFER_READ_BIT |
                   VK_ACCESS_TRANSFER_WRITE_BIT |
                   VK_ACCESS_HOST_READ_BIT |
                   VK_ACCESS_HOST_WRITE_BIT};

vkCmdPipelineBarrier(
    ...
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT, // srcStageMask
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT, // dstStageMask
    1,                                  // memoryBarrierCount
    &memoryBarrier,                     // pMemoryBarriers
    ...);
```

TODO

- Dynamic upload examples (e.g. per-frame uniforms)
- Transfer read-back examples