Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

Queue

Logical Device

Queue

Logical Device

Queue

Logical Device

Queue

Logical Device

Queue

Logical Device

Command Buffer

Queue

Command Buffer

Queue

Command Buffer
Simplified Block Diagram

- Application
- Instance
- Physical Device
- Logical Device
- Queue
  - Command Buffer
  - Command Buffer
  - Command Buffer
Vulkan Queues and Command Buffers

- Graphics commands are recorded in command buffers, e.g., `vkCmdDoSomething(cmdBuffer, ...)``;
- You can have as many simultaneous Command Buffers as you want;
- Each command buffer can be filled from a different thread, but doesn't have to be;
- Command Buffers record commands, but no work takes place until a Command Buffer is submitted to a Queue;
- We don't create Queues – the Logical Device already has them;
- Each Queue belongs to a Queue Family;
- We don't create Queue Families – the Physical Device already has them.
Querying what Queue Families are Available

uint32_t count;
VkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr );

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceFamilyProperties( PhysicalDevice, &count, OUT &vqfp, );

for( unsigned int i = 0; i < count; i++ )
{
    fprintf( FpDebug, "\t%d: Queue Family Count = %2d ; ", i, vqfp[i].queueCount );
    if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )    fprintf( FpDebug, " Graphics" );
    if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT ) != 0 )       fprintf( FpDebug, " Compute " );
    if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )       fprintf( FpDebug, " Transfer" );
    fprintf(FpDebug, "\n");
}

For the Nvidia A6000 cards:

<table>
<thead>
<tr>
<th>Queue Family Count</th>
<th>Graphics</th>
<th>Compute</th>
<th>Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Found 3 Queue Families:
Similarly, we Can Write a Function that Finds the Proper Queue Family

```c
int FindQueueFamilyThatDoesGraphics( )
{
    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr );

    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, IN &count, OUT vqfp );

    for( unsigned int i = 0; i < count; i++ )
    {
        if( ( vqfp[ i ].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
            return i;
    }
    return -1;
}
```
Creating a Logical Device Needs to Know Queue Family Information

```c
float queuePriorities[ ] =
{
    1. // one entry per queueCount
};

VkDeviceQueueCreateInfo vdqci[1];
    vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
    vdqci[0].pNext = nullptr;
    vdqci[0].flags = 0;
    vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
    vdqci[0].queueCount = 1;
    vdqci[0].queuePriorities = (float *) queuePriorities;

VkDeviceCreateInfo vdci;
    vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
    vdci.pNext = nullptr;
    vdci.flags = 0;
    vdci.queueCreateInfoCount = 1; // # of device queues wanted
    vdci.pQueueCreateInfos = &vdqci[0]; // array of VkDeviceQueueCreateInfo's
    vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
    vdci.ppEnabledLayerNames = myDeviceLayers;
    vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
    vdci.ppEnabledExtensionNames = myDeviceExtensions;
    vdci.pEnabledFeatures = &PhysicalDeviceFeatures; // already created

result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );

VkQueue Queue;
    uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
    uint32_t queueIndex = 0;

result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
```
Creating the Command Pool as part of the Logical Device

```c
VkResult
Init06CommandPool( )
{
    VkResult result;

    VkCommandPoolCreateInfo vcpci;
    vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcpci.pNext = nullptr;
    vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT |
                  VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;

#ifdef CHOICES
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
#endif
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );

    result = vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool );

    return result;
}
```
Creating the Command Buffers

```c
VkResult
Init06CommandBuffers()
{
    VkResult result;

    // allocate 2 command buffers for the double-buffered rendering:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 2;           // 2, because of double-buffering
        result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
    }

    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    {
        VkCommandBufferAllocateInfo vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = CommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 1;
        result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );
    }

    return result;
}
```
Beginning a Command Buffer – One per Image

```c
VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX,
                           IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );

VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

    result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );

    . . .

    vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```
Beginning a Command Buffer

- `vkBeginCommandBuffer()`
- `VkCommandBufferBeginInfo`
- `vkAllocateCommandBuffer()`
- `VkCommandBufferAllocateInfo`
- `vkCreateCommandBufferPool()`
- `VkCommandBufferPoolCreateInfo`
These are the Commands that could be entered into a Command Buffer, I

vkCmdBeginConditionalRendering
vkCmdBeginDebugUtilsLabel
vkCmdBeginQuery
vkCmdBeginQueryIndexed
vkCmdBeginRendering
vkCmdBeginRenderPass
vkCmdBeginRenderPass2
vkCmdBeginTransformFeedback
vkCmdBindDescriptorSets
vkCmdBindIndexBuffer
vkCmdBindInvocationMask
vkCmdBindPipeline
vkCmdBindPipelineShaderGroup
vkCmdBindShadingRateImage
vkCmdBindTransformFeedbackBuffers
vkCmdBindVertexBuffers
vkCmdBindVertexBuffers2
vkCmdBlitImage
vkCmdBlitImage2
vkCmdBuildAccelerationStructure
vkCmdBuildAccelerationStructuresIndirect
vkCmdBuildAccelerationStructures
vkCmdClearAttachments
vkCmdClearColorImage
vkCmdClearDepthStencilImage
vkCmdCopyAccelerationStructure
vkCmdCopyAccelerationStructureToMemory
vkCmdCopyBuffer
vkCmdCopyBuffer2
vkCmdCopyBufferToImage
vkCmdCopyBufferToImage2
vkCmdCopyImage
vkCmdCopyImage2
vkCmdCopyImageToBuffer
vkCmdCopyImageToBuffer2
vkCmdCopyMemoryToAccelerationStructure
### These are the Commands that could be entered into a Command Buffer, II

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCmdCopyQueryPoolResults</td>
<td>vkCmdDrawMulti</td>
</tr>
<tr>
<td>vkCmdCuLaunchKernelX</td>
<td>vkCmdDrawMultiIndexed</td>
</tr>
<tr>
<td>vkCmdDebugMarkerBegin</td>
<td>vkCmdEndConditionalRendering</td>
</tr>
<tr>
<td>vkCmdDebugMarkerEnd</td>
<td>vkCmdEndDebugUtilsLabel</td>
</tr>
<tr>
<td>vkCmdDebugMarkerInsert</td>
<td>vkCmdEndQuery</td>
</tr>
<tr>
<td>vkCmdDispatch</td>
<td>vkCmdEndQueryIndexed</td>
</tr>
<tr>
<td>vkCmdDispatchBase</td>
<td>vkCmdEndRendering</td>
</tr>
<tr>
<td>vkCmdDispatchIndirect</td>
<td>vkCmdEndRenderPass</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectByteCount</td>
<td>vkCmdEndRenderPass2</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCount</td>
<td>vkCmdEndTransformFeedback</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCountByteCount</td>
<td>vkCmdExecuteCommands</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCount</td>
<td>vkCmdExecuteGeneratedCommands</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCountByteCount</td>
<td>vkCmdFillBuffer</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCount</td>
<td>vkCmdInsertDebugUtilsLabel</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectByteCountCount</td>
<td>vkCmdNextSubpass</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCount</td>
<td>vkCmdNextSubpass2</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCountByteCountCount</td>
<td>vkCmdPipelineBarrier</td>
</tr>
<tr>
<td>vkCmdDispatchIndirectCountByteCountCount</td>
<td>vkCmdPipelineBarrier2</td>
</tr>
<tr>
<td>vkCmdDraw</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndexed</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirect</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndexedIndirectCount</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndirect</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndirectByteCount</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawIndirectCount</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirectCount</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirectCountByteCount</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawMeshTasksIndirect</td>
<td></td>
</tr>
<tr>
<td>vkCmdDrawMeshTasks</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndConditionalRendering</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndDebugUtilsLabel</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndQuery</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndQueryIndexed</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndRendering</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndRenderPass</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndRenderPass2</td>
<td></td>
</tr>
<tr>
<td>vkCmdEndTransformFeedback</td>
<td></td>
</tr>
<tr>
<td>vkCmdExecuteCommands</td>
<td></td>
</tr>
<tr>
<td>vkCmdExecuteGeneratedCommands</td>
<td></td>
</tr>
<tr>
<td>vkCmdFillBuffer</td>
<td></td>
</tr>
<tr>
<td>vkCmdInsertDebugUtilsLabel</td>
<td></td>
</tr>
<tr>
<td>vkCmdNextSubpass</td>
<td></td>
</tr>
<tr>
<td>vkCmdNextSubpass2</td>
<td></td>
</tr>
<tr>
<td>vkCmdPipelineBarrier</td>
<td></td>
</tr>
<tr>
<td>vkCmdPipelineBarrier2</td>
<td></td>
</tr>
</tbody>
</table>
These are the Commands that could be entered into a Command Buffer, III

vkCmdPreprocessGeneratedCommands  vkCmdSetDepthTestEnable
vkCmdPushConstants                  vkCmdSetDepthWriteEnable
vkCmdPushDescriptorSet              vkCmdSetDeviceMask
vkCmdPushDescriptorSetWithTemplate  vkCmdSetDiscardRectangle
vkCmdResetEvent                     vkCmdSetEvent
vkCmdResetEvent2                    vkCmdSetEvent2
vkCmdResetQueryPool                 vkCmdSetExclusiveScissor
vkCmdResolveImage                   vkCmdSetFragmentShadingRateEnum
vkCmdResolveImage2                  vkCmdSetFragmentShadingRate
vkCmdSetBlendConstants              vkCmdSetFrontFace
vkCmdSetCheckpoint                   vkCmdSetLineStipple
vkCmdSetCoarseSampleOrder           vkCmdSetLineWidth
vkCmdSetCullMode                     vkCmdSetLogicOp
vkCmdSetDepthBias                   vkCmdSetPatchControlPoints
vkCmdSetDepthBiasEnable             vkCmdSetPrimitiveRestartEnable
vkCmdSetDepthBounds                 vkCmdSetPrimitiveTopology
vkCmdSetDepthBoundsTestEnable       vkCmdSetRasterizerDiscardEnable
vkCmdSetDepthCompareOp              vkCmdSetRayTracingPipelineStackSize
These are the Commands that could be entered into a Command Buffer, IV

```
vkCmdSetSampleLocations
vkCmdSetScissor
vkCmdSetScissorWithCount
vkCmdSetStencilCompareMask
vkCmdSetStencilOp
vkCmdSetStencilReference
vkCmdSetStencilTestEnable
vkCmdSetStencilWriteMask
vkCmdSetVertexInput
vkCmdSetViewport
vkCmdSetViewportShadingRatePalette
vkCmdSetViewportWithCount
vkCmdSetViewportWScaling
```

```
vkCmdSubpassShading
vkCmdTraceRaysIndirect2
vkCmdTraceRaysIndirect
vkCmdTraceRays
vkCmdUpdateBuffer
vkCmdWaitEvents
vkCmdWaitEvents2
vkCmdWriteAccelerationStructuresProperties
vkCmdWriteBufferMarker2
vkCmdWriteBufferMarker
vkCmdWriteTimestamp
vkCmdWriteTimestamp2
```
How the `RenderScene()` Function Works

```c
VkResult
RenderScene( )
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

    uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX, IN VK_NULL_HANDLE, IN VK_NULL_HANDLE, OUT &nextImageIndex );

    VkCommandBufferBeginInfo vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
    result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
```
VkClearColorValue
vccv.float32[0] = 0.0;
vccv.float32[1] = 0.0;
vccv.float32[2] = 0.0;
vccv.float32[3] = 1.0;

VkClearDepthStencilValue
vcdsv.depth = 1.f;
vcdsv.stencil = 0;

VkClearValue
vcv[0].color = vccv;
vcv[1].depthStencil = vcdsv;

VkOffset2D o2d = { 0, 0 };
VkExtent2D e2d = { Width, Height };
VkRect2D r2d = { o2d, e2d };

VkRenderPassBeginInfo
vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
vrpbi.pNext = nullptr;
vrpbi.renderPass = RenderPass;
vrpbi.framebuffer = Framebuffers[nextImageIndex];
vrpbi.renderArea = r2d;
vrpbi.clearValueCount = 2;
vrpbi.pClearValues = vcv; // used for VK_ATTACHMENT_LOAD_OP_CLEAR

vkCmdBeginRenderPass(CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE);
VkViewport viewport =
{
    0.,                     // x
    0.,                     // y
    (float)Width,
    (float)Height,
    0.,                     // minDepth
    1.                      // maxDepth
};

vkCmdSetViewport( CommandBuffers[nextImageIndex], 0, 1, IN &viewport );         // 0=firstViewport, 1=viewportCount

VkRect2D scissor =
{
    0,
    0,
    Width,
    Height
};

vkCmdSetScissor( CommandBuffers[nextImageIndex], 0, 1, IN &scissor );

vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
    GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
    // dynamic offset count, dynamic offsets

vkCmdBindPushConstants( CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values );

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };

VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );               // 0, 1 = firstBinding, bindingCount

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
Submitting a Command Buffer to a Queue for Execution

```c
VkSubmitInfo vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &CommandBuffer;
    vsi.waitSemaphoreCount = 1;
    vsi.pWaitSemaphores = imageReadySemaphore;
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
    vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
```
The Entire Submission / Wait / Display Process

```
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;

VkFence renderFence;
vkCreateFence( LogicalDevice, IN &vfci, PALLOCATOR, OUT &renderFence );
result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue );
// 0 = queueIndex

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsipNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsicommandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsisignalSemaphoreCount = 0;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;

result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence );  // 1 = submitCount
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );  // waitAll, timeout

vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );

VkPresentInfoKHR vpi;
vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
vpi.pNext = nullptr;
vpi.waitSemaphoreCount = 0;
vpi.pWaitSemaphores = nullptr;
vpi.swapchainCount = 1;
vpi.pSwapchains = &SwapChain;
vpi.pImageIndices = &nextImageIndex;
vpi.pResults = nullptr;

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```
What Happens After a Queue has Been Submitted?

As the Vulkan Specification says:

“Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences.”

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there.

But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.