Queues and Command Buffers

Application

Instance

Physical Device

Logical Device

Command Buffer

Vulkan: Overall Block Diagram

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

CPU Thread

CPU Thread

CPU Thread

CPU Thread

Cmd buffer

Cmd buffer

Cmd buffer

Cmd buffer

queue

queue

queue

queue

Querying what Queue Families are Available

uint32_t count;
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++ )
{
fprintf(FpDebug, "\t%d: Queue Family Count = %2d  ;   ", i, vqfp[i].queueCount);
if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
printf( FpDebug, " Graphics\n");
if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT ) != 0 )
printf( FpDebug, " Compute\n");
if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )
printf( FpDebug, " Transfer\n");
fclose(FpDebug);
}

For the Nvidia A6000 cards:

Found 3 Queue Families:

0: Queue Family Count = 16  ;   Graphics Compute Transfer
1: Queue Family Count =   2  ;   Transfer
2: Queue Family Count =   8  ;   Compute Transfer

Similarly, we Can Write a Function that Finds the Proper Queue Family

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

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

Creating the Command Pool as part of the Logical Device

```cpp
VkCommandPoolCreateInfo vcpci;
vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
```

Beginning a Command Buffer – One per Image

```cpp
vkBeginCommandBuffer( CommandBuffers[ nextImageIndex ];
```

These are the Commands that could be entered into a Command Buffer, 1

- `vkCmdDrawIndexed` (draws indexed vertices)
- `vkCmdDrawIndirect` (draws indexed vertices using buffer)
- `vkCmdDrawArraysIndirect` (draws arrays using buffer)
- `vkCmdDispatch` (sends work to threads)
- `vkCmdDispatchIndirect` (sends work to threads using buffer)
- `vkCmdSetViewport` (sets viewport)
- `vkCmdSetScissor` (sets scissor)
- `vkCmdSetLineWidth` (sets line width)

---

**Image Descriptions**
- Creating a Logical Device Needs to Know Queue Family Information
- Creating the Command Pool as part of the Logical Device
- Beginning a Command Buffer – One per Image
- These are the Commands that could be entered into a Command Buffer, 1
Submitting a Command Buffer to a Queue for Execution

```cpp
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 = nullptr;
vsi.pWaitDstStageMask = nullptr;
```

Submitting a Command Buffer to a Queue for Execution

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

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

VkQueue presentQueue;
result = vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, &presentQueue);

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;
```

The Entire Submission / Wait / Display Process

```cpp
result = vkQueueSubmit(presentQueue, 1, &vsi, renderFence); // 1 = submitCount
result = vkWaitForFences(LogicalDevice, 1, &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;
```

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.