Keeping your GPU fed without getting bitten

Tobias Hector
May 2016
Introduction

- You have delicious draw calls
  - Yummy!
Introduction

- You have delicious draw calls
  - Yummy!

- Your GPU wants to eat them
  - It’s really hungry
Introduction

- You have delicious draw calls
  - Yummy!

- Your GPU wants to eat them
  - It’s really hungry

- Keep it fed at all times
  - So it keeps making pixels
Introduction

• You have delicious draw calls
  - Yummy!

• Your GPU wants to eat them
  - It’s really hungry

• Keep it fed at all times
  - So it keeps making pixels

• Don’t want it biting your hand
  - Look at those teeth!
Keeping it fed

- GPU needs a constant supply of food
  - It doesn’t want to wait

- Certain foods are tough to digest
  - Provide multiple operations to hide stalls

- Draw calls provide a variety of nutrition
  - Vertex work, raster work, tessellation, vitamins A-K, etc.
# Keeping it fed

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## Keeping it fed

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
## Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>Fragment</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
## Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex</strong></td>
<td>0</td>
</tr>
<tr>
<td><strong>Fragment</strong></td>
<td>0</td>
</tr>
</tbody>
</table>
Not getting bitten

• GPU eating from lots of different plates
  - Don’t touch anything it’s using!

• It doesn’t want a mouthful of beef choc chip ice cream
  - Don’t change data whilst it’s accessing a resource

• Hey I’m eating that!
  - Don’t delete resources whilst the GPU is still using them
On to the serious bits...
Terminology

- **Operation**
  - Anything that can be executed
  - Includes synchronization and memory barriers

- **Execution Dependency**
  - Operations waiting on other operations
  - All synchronization expresses these

- **Memory Barrier**
  - Flush/invalidate caches
  - Determination of access and visibility

- **Memory Dependency**
  - Execution dependency involving a Memory Barrier

Note: Memory barrier does not mean quite the same thing as GL’s memory barrier, though there is some relation.
Synchronization Types

- 3 types of explicit synchronization in Vulkan

- Pipeline Barriers, Events and Subpass Dependencies
  - Within a queue
  - Explicit memory dependencies

- Semaphores
  - Between Queues

- Fences
  - Whole queue operations to CPU

OpenGL has just two, very coarse synchronization primitives: memory barriers and fences. They are loosely similar to the equivalently named concepts in Vulkan.
Pipeline Barriers

- Pipeline Barriers
  - Precise set of pipeline stages
  - Memory Barriers to execute
  - Single point in time

Executing a pipeline barrier is roughly equivalent to a glMemoryBarrier call, though with much more control.

```c
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);
```
**Events**

- **Events**
  - Same info as Pipeline Barriers
  - ...but operate over a range

```c
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);

void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);

void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```
Events

- **Events**
  - Same info as Pipeline Barriers
  - ...but operate over a range

- **CPU interaction**
  - No explicit CPU wait
  - No Memory Barriers

```c
VkResult vkSetEvent(
    VkDevice device,
    VkEvent event);

VkResult vkResetEvent(
    VkDevice device,
    VkEvent event);

VkResult vkGetEventStatus(
    VkDevice device,
    VkEvent event);
```
Events

• **Events**
  - Same info as Pipeline Barriers
  - …but operate over a range

• **CPU interaction**
  - No explicit CPU wait
  - No Memory Barriers

• **Warning!**
  - OS may apply a timeout
  - Set events soon after submission
  - Could you just defer submission?
Pipeline Barriers vs Events

- Use pipeline barriers for point synchronization
  - Dependant operation immediately precedes operation that depends on it
  - May be more optimal than set/wait event pair

- Use events if other work possible between two operations
  - Set immediately after the dependant operation
  - Wait immediately before the operation that depends on it

- Use events for CPU/GPU synchronization
  - Memory accesses between processors
  - Late latching of data to reduce latency
Memory Barrier Types

- **Global Memory Barrier**
  - All memory-backed resources

- **Buffer Barrier**
  - For a single buffer range

- **Image Barrier**
  - For a single image subresource range

OpenGL’s memory barriers imply execution dependencies, which Vulkan memory barriers do not - execution barriers are provided by a pipeline barrier, event or subpass dependency.
Global Memory Barriers

- Global Memory Barriers
  - All memory used by accessed stages
  - Effectively flushes entire caches

- Use when many resources transition
  - Cheaper than one-by-one
  - Don’t transition unnecessarily!

- User must define prior access
  - Driver not tracking for you

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

- A single buffer range
- Defines access stages
- Defines queue ownership

User must define prior access
- Driver not tracking for you

```c
typedef struct VkBufferMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier;
```
Image Barriers

• Image Barriers
  - A single image subresource range
  - Defines access stages
  - Defines queue ownership
  - Defines image layout

• User must define prior access
  - Driver not tracking for you
  - For images, this includes prior layout

• Appropriate layouts allow compression
  - GPU may use image compression
  - Saves bandwidth
  - Use GENERAL instead of switching frequently

```c
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;
```
Subpass Dependencies

- **Subpass dependencies**
  - Similar info to Pipeline Barriers
  - Explicitly between two subpasses

- **Memory barriers**
  - Implicit for attachments
  - Explicit for other resources

- **Pixel local dependencies**
  - Same fragment/sample location
  - Cheap for most implementations
  - Use region dependency flag:
    - `VK_DEPENDENCY_BY_REGION_BIT`

```c
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;
```
Subpass Dependencies

- **Subpass self-dependencies**
  - Subpasses can wait on themselves
  - A pipeline barrier in the subpass

- **Forward progress only**
  - Can’t wait on later stages
  - Must wait on earlier or same stage

- **Pixel local only between fragments**
  - Must use flag:
    - VK_DEPENDENCY_BY_REGION_BIT

```c
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;

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);
```
Subpass Dependencies

- Subpass external dependencies
  - Wait on ‘external’ operations
  - vkCmdWaitEvent in the subpass
  - Events set outside the render pass

```c
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;

void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```
Example - Texture Upload

// Transition the buffer from host write to transfer read
bufferBarrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
bufferBarrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;

// Transition the image to transfer destination
imageBarrier.srcAccessMask = 0;
imageBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
imageBarrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
imageBarrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, &bufferBarrier, &imageBarrier);

vkCmdCopyBufferToImage(commandBuffer, srcBuffer, image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &copy);

// Transition the image from transfer destination to shader read
imageBarrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
imageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
imageBarrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
imageBarrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, &imageBarrier);

Example - Compute to Draw Indirect

// Add a subpass dependency to express the wait on an external event
define externalDependency
  .srcSubpass = VK_SUBPASS_EXTERNAL;
define externalDependency
  .srcStageMask = VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT;
define externalDependency
  .dstStageMask = VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT;
define externalDependency
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
define externalDependency
  .dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;

// Dispatch a compute shader that generates indirect command structures
vkCmdDispatch();

// Set an event that can be later waited on (same source stage).
vkCmdSetEvent(commandBuffer, event, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT);

vkCmdBeginRenderPass(...);

// Transition the buffer from shader write to indirect command
bufferBarrier
  .srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
define bufferBarrier
  .dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;
define bufferBarrier
  .buffer = indirectBuffer;
vkCmdWaitEvent(commandBuffer, event, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT, &bufferBarrier);

vkCmdDrawIndirect(commandBuffer, indirectBuffer, ...);
Semaphores

- **Semaphores**
  - Used to synchronize queues
  - Not necessary for single-queue

- **Fairly coarse grain**
  - Per submission batch
    - E.g. a set of command buffers
  - Multiple per submit command

- **Implicit memory guarantees**
  - Effects visible to future operations on the same device
  - Not guaranteed visible to host

```c
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t commandBufferCount;
    const VkCommandBuffer* pCommandBuffers;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkSubmitInfo;
```
Example - Acquire and Present

// Acquire an image. Pass in a semaphore to be signalled
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, acquireSemaphore, VK_NULL_HANDLE, &imageIndex);

// Submit command buffers
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &acquireSemaphore;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;
submitInfo.signalSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &graphicsSemaphore;

vkQueueSubmit(graphicsQueue, 1, &submitInfo, fence);

// Present images to the display
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &graphicsSemaphore;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapchain;
presentInfo.pImageIndices = &imageIndex;

vkQueuePresent(presentQueue, &presentInfo);
Example - Acquire and Present (same queue)

```c
// Acquire an image. Pass in a semaphore to be signalled
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, acquireSemaphore, VK_NULL_HANDLE, &imageIndex);

// Submit command buffers
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &acquireSemaphore;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;
submitInfo.signalSemaphoreCount = 0;

vkQueueSubmit(universalQueue, 1, &submitInfo, fence);

// Present images to the display
presentInfo.waitSemaphoreCount = 0;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapchain;
presentInfo.pImageIndices = &imageIndex;

vkQueuePresent(universalQueue, &presentInfo);
```
Fences

- **Fences**
  - Used to synchronize queue to CPU

- **Very coarse grain**
  - Per queue submit command

- **Implicit memory guarantees**
  - Effects visible to future operations on the same device
  - Not guaranteed visible to host

GL's fences are like a combination of a semaphore and a fence in Vulkan - they can synchronize GPU and CPU in multiple ways at a coarse granularity.
Example - Multi-buffering

// Have enough resources and fences to have one per in-flight-frame, usually the swapchain image count
VkBuffer buffers[swapchainImageCount];
VkFence fence[swapchainImageCount];

// Can use the index from the presentation engine - 1:1 mapping between swapchain images and resources
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, semaphore, VK_NULL_HANDLE, &nextIndex);

// Make absolutely sure that the work has completed
vkWaitForFences(device, 1, &fence[nextIndex], true, UINT64_MAX);

// Reset the fences we waited on, so they can be re-used
vkResetFences(device, 1, &fence[nextIndex]);

// Change the data in your per-frame resources (with appropriate events/barriers!)
...

// Submit any work to the queue, with those fences being re-used for the next time around
vkQueueSubmit(graphicsQueue, 1, &sSubmitInfo, fence[nextIndex]);
Wait Idle

- Ensures execution completes
  - VERY heavy-weight

- vkQueueWaitIdle
  - Wait for queue operations to finish
  - Equivalent to waiting on a fence

- vkDeviceWaitIdle
  - Waits for device operations to finish
  - Includes vkQueueWaitIdle for queues

These are a lot like glFinish, and should be treated similarly - use them VERY SPARINGLY.
Wait Idle

- Useful primarily at teardown
  - Use it to quickly ensure all work is done

- Favour other synchronization at all other times
  - Extremely heavyweight, will cause serialization!
Programmer Guidelines

- Specify EXACTLY the right amount of synchronization
  - Too much and you risk starving your GPU
  - Miss any and your GPU will bite you

- Use the validation layers to help!
  - Won’t catch everything yet, but improving over time

- Pay particular attention to the pipeline stages
  - Fiddly but become intuitive as you use them

- Consider Image Layouts
  - If your GPU can save bandwidth it will

- Different behaviour depending on implementation
  - Test/Tune on every platform you can find!
Keep your GPU fed without getting bitten!

Questions?