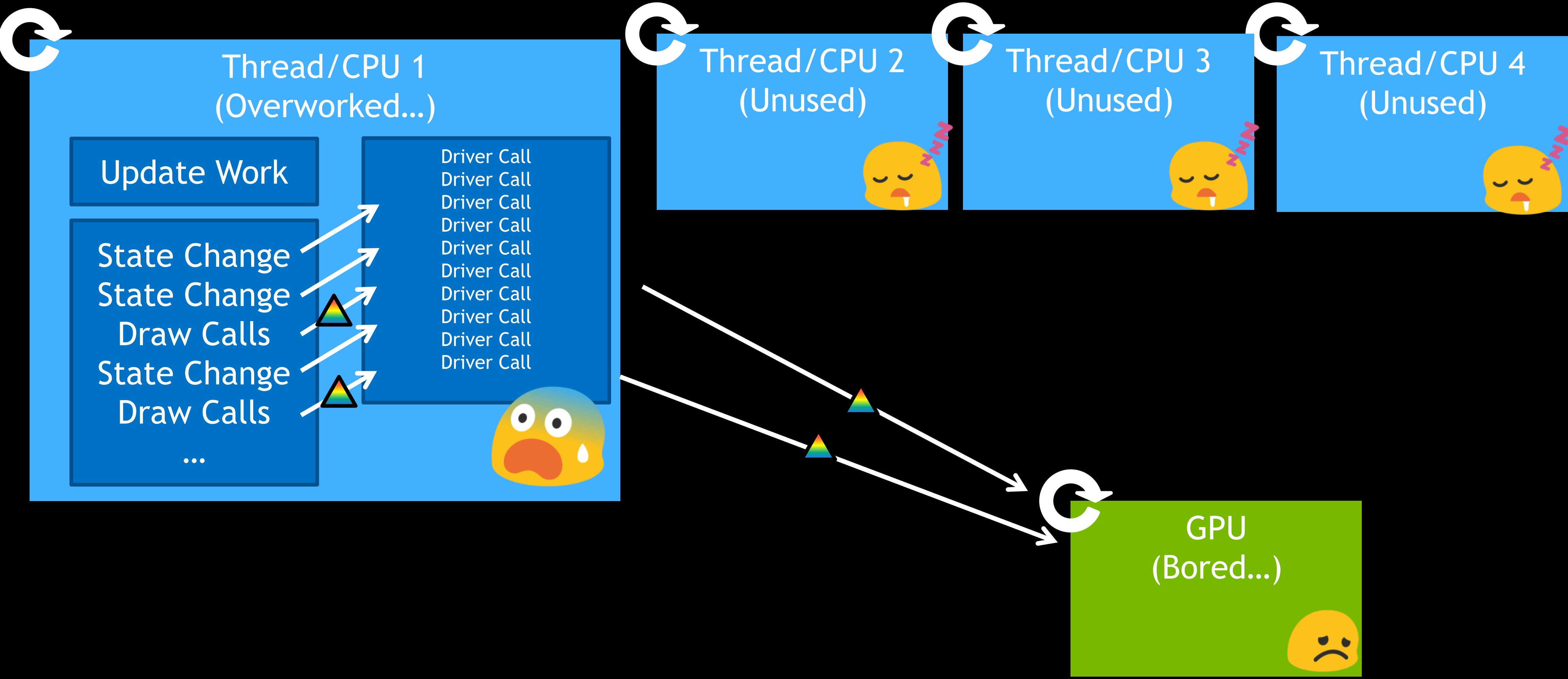
Vulkan Multi-Threading

Khronos Munich Chapter meeting, April 8th 2016

Mathias Schott, Senior Developer Technology Engineer, NVIDIA

What is the issue?



Thread/CPU 1
(Overworked...)

Update Work

State Change
State Change
Draw Calls
State Change
Draw Calls
...

Driver Call
Driver Call
Driver Call
Driver Call
Driver Call
Driver Call
Driver Call
Driver Call
Driver Call
Driver Call

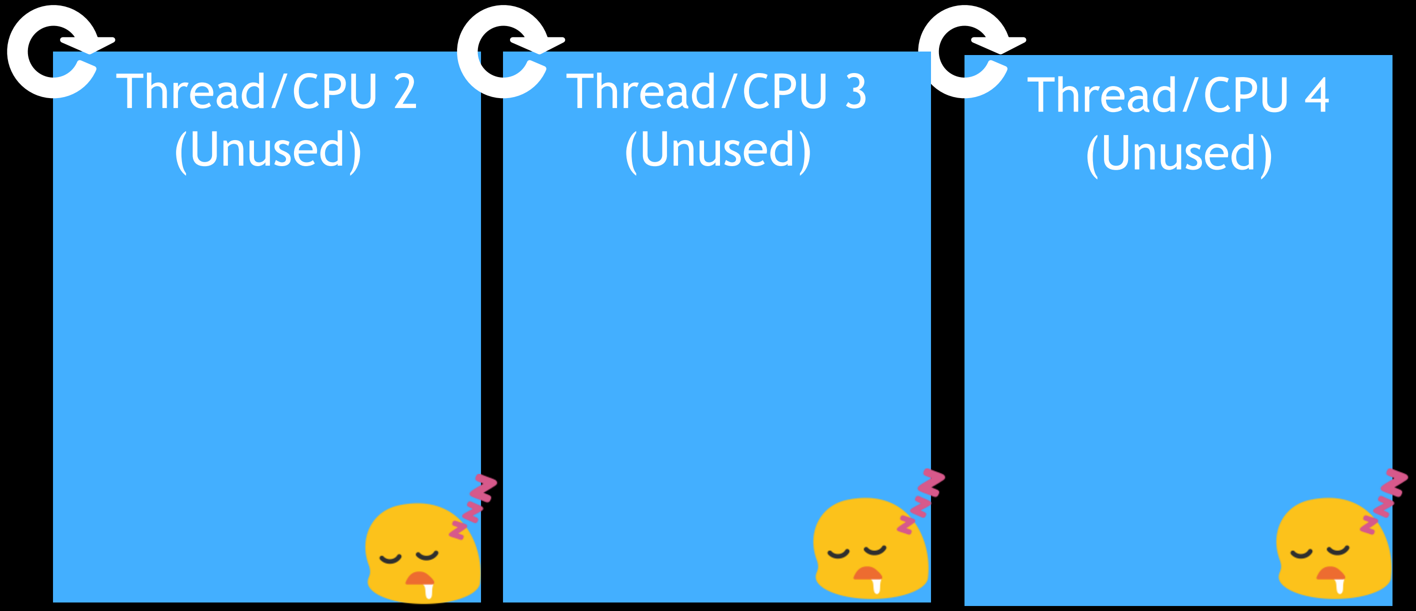
Thread/CPU 2
(Unused)

Thread/CPU 3
(Unused)

Thread/CPU 4
(Unused)

GPU
(Bored...)

Developers Want Threading-Friendly APIs!

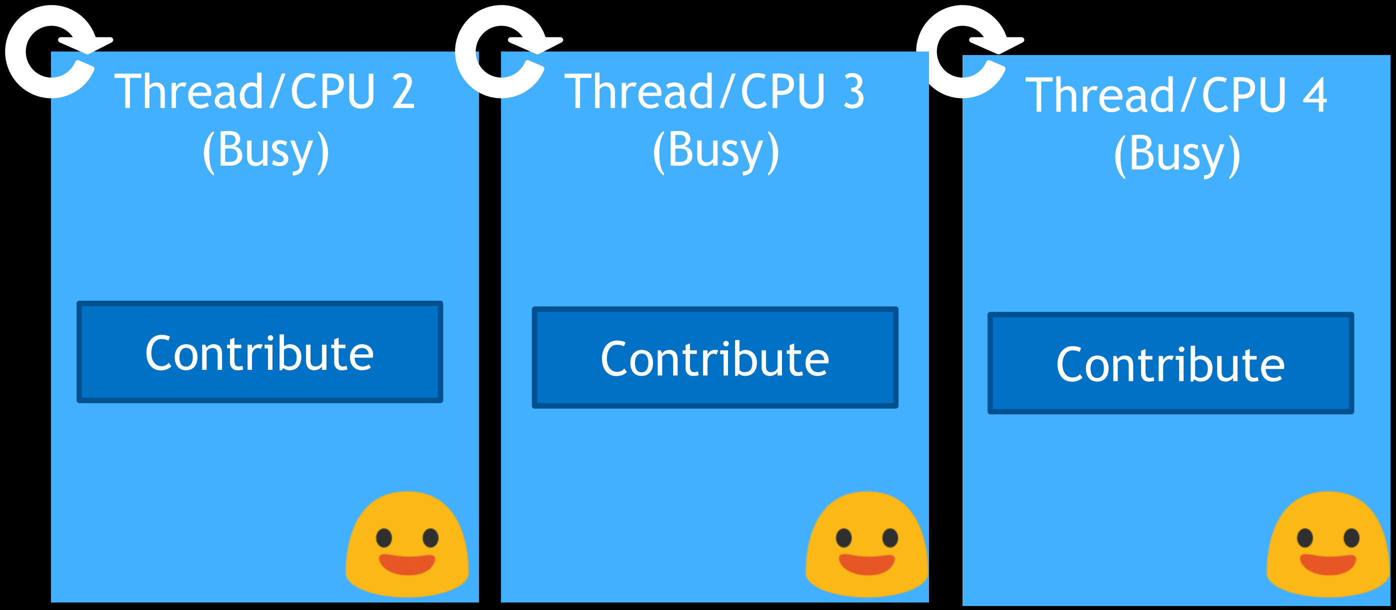


Thread/CPU 2
(Unused)

Thread/CPU 3
(Unused)

Thread/CPU 4
(Unused)

Developers Want Threading-Friendly APIs!



Thread/CPU 2 (Busy)

Contribute

Thread/CPU 3 (Busy)

Contribute

Thread/CPU 4 (Busy)

Contribute

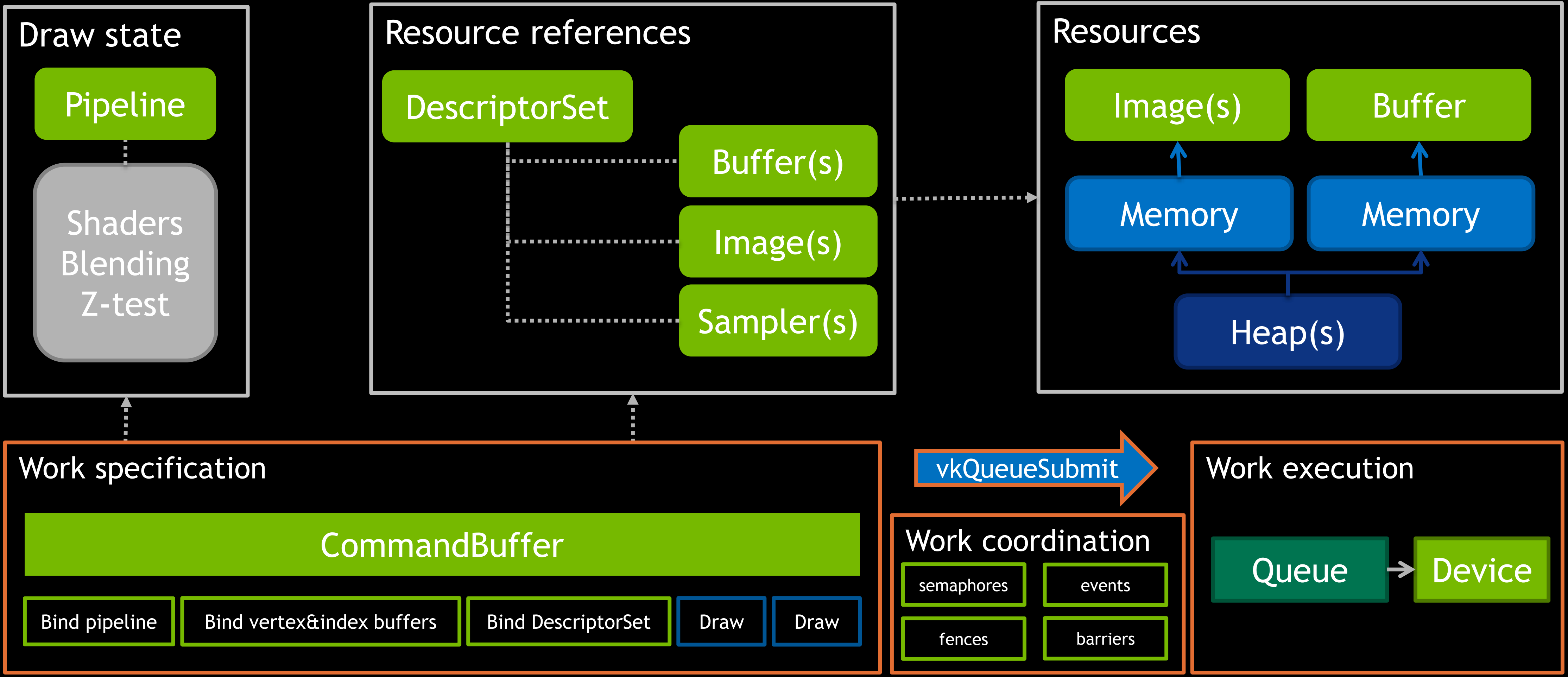
Vulkan Philosophy: Explicit Threadability

- Vulkan was created from the ground up to be thread-friendly
  - A huge amount of the spec details the thread-safety and consequences of calls
  - But all of the responsibility falls on the app – which is good!
- Threading at the app level continues to rise in popularity
  - Apps want to generate rendering work from multiple threads
  - Spread validation and submission costs across multiple threads
  - Apps can often handle object/access synchronization at a higher level than a driver

Threading use cases encouraged in Vulkan

- 🤐 Threaded updates of resources (Buffers)
  - CPU vertex data or instance data animations (e.g. morphing)
  - CPU uniform buffer data updates (e.g. transform updates)
- 🤐 Parallel pipeline state creation
  - “shader compilation” and state validation
- 🕵️ Threaded rendering / draw calls
  - Generation of command buffers in multiple threads

Separate work specification & submission!



https://developer.nvidia.com/vulkan

Work Specification: Command Buffers

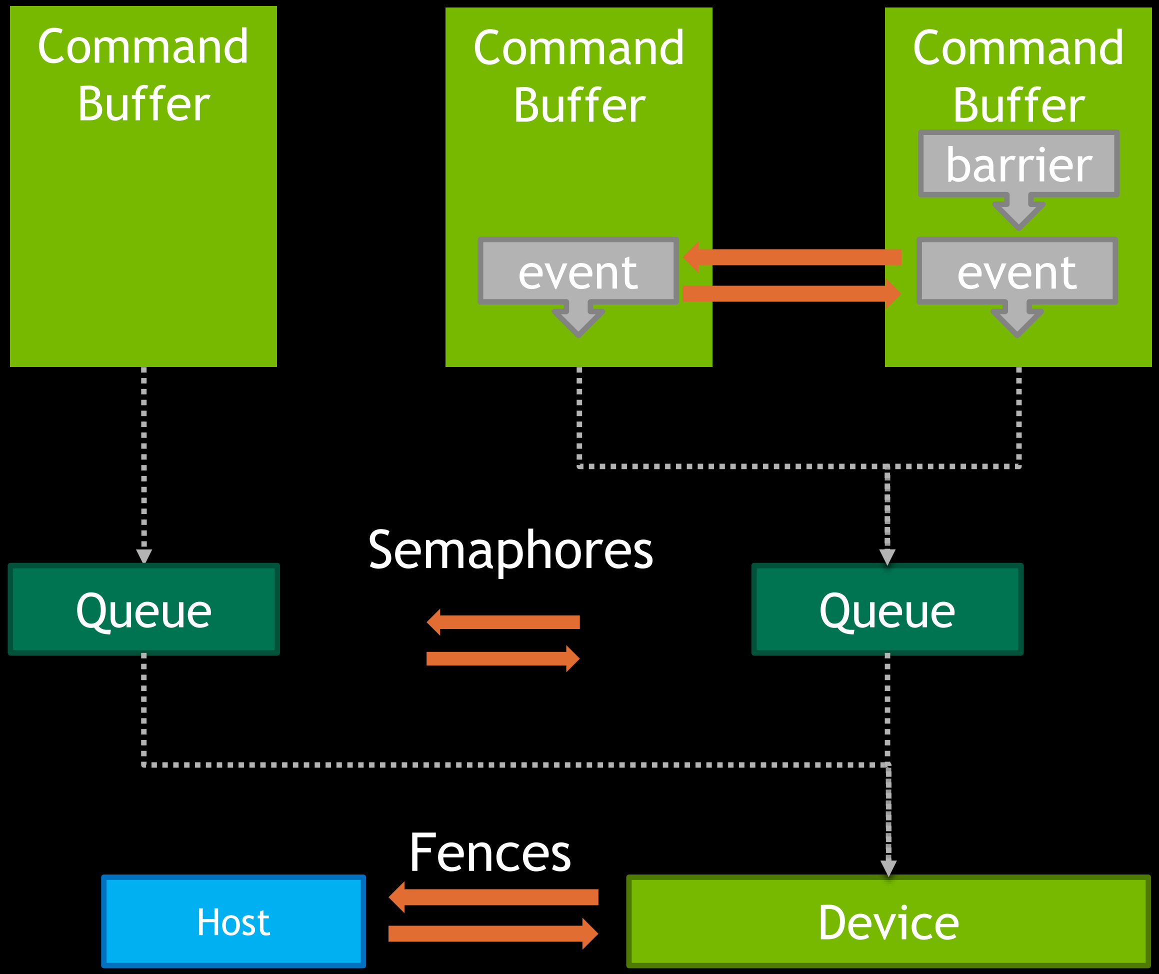
- All Vulkan rendering is through command buffers
- Can be single-use or multi-submission
  - Driver can optimize the buffer accordingly
- Primary & Secondary Command buffers
  - Allow static work to be reused
- *IMPORTANT: No state is inherited across command buffers!*

Work Execution: Queues

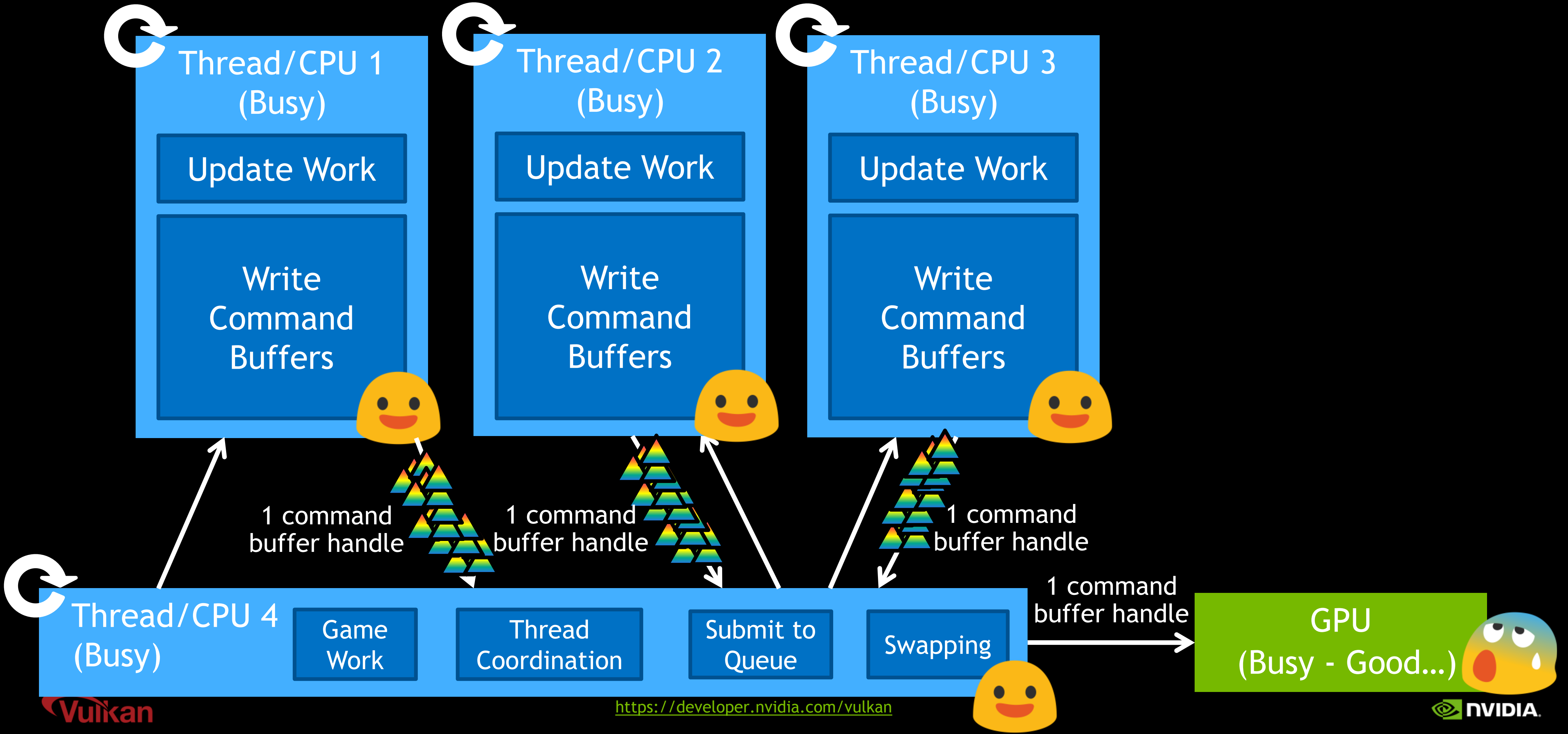
- Makes explicit the command queue that is implicitly in a context in GL
  - No need to "bind a context" in order to submit work
  - Multiple threads can submit work to different queues
- Queues accept GPU work via CommandBuffer submissions
  - Queues have extremely few operations: in essence, "submit work" and "wait for idle"
- Queue work submissions can include sync primitives for the queue to:
  - ***Wait*** upon before processing the submitted work
  - ***Signal*** when the work in this submission is completed
- Queue "families" can accept different types of work, e.g.
    - One form of work in a queue (e.g. DMA/memory transfer-only queue)

Work Coordination: Synchronization

- **semaphores**
  - used to synchronize work across queues or across coarse-grained submissions to a single queue
- **events** and **barriers**
  - used to synchronize work within a command buffer or sequence of command buffers submitted to a single queue
- **fences**
  - used to synchronize work between the device and the host.



Threaded Command Buffer Generation



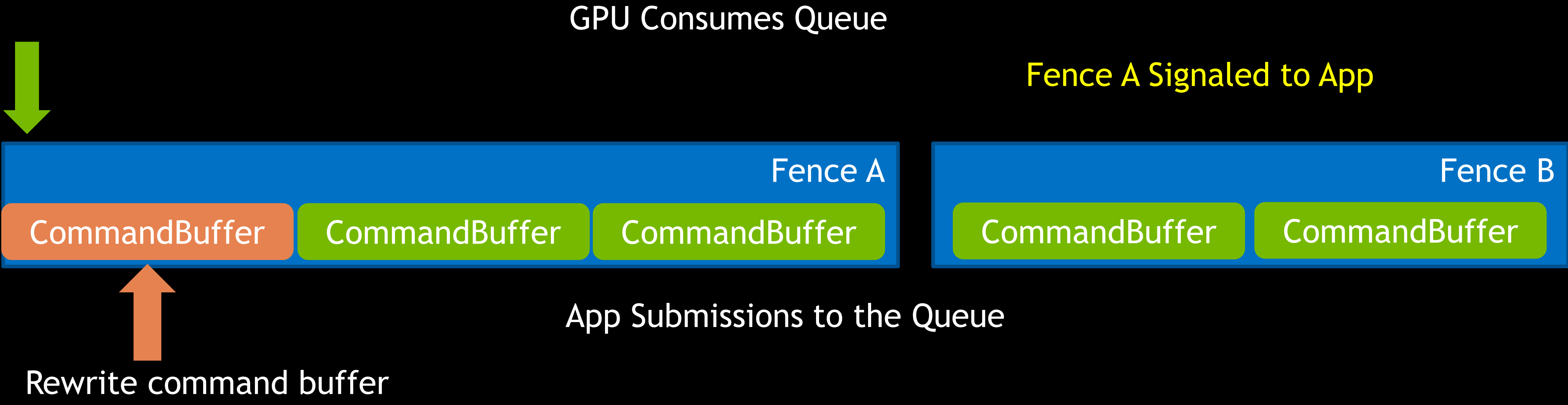
Vulkan

https://developer.nvidia.com/vulkan

NVIDIA

Command Buffer Thread Safety

- Must not recycle a CommandBuffer for rewriting until it is no longer in flight
- But we do not want to flush the queue each frame!
- VkFences can be provided with a queue submission to test when a command buffer is ready to be recycled



GPU Consumes Queue

Fence A Signaled to App

Fence A

CommandBuffer

CommandBuffer

CommandBuffer

Fence B

CommandBuffer

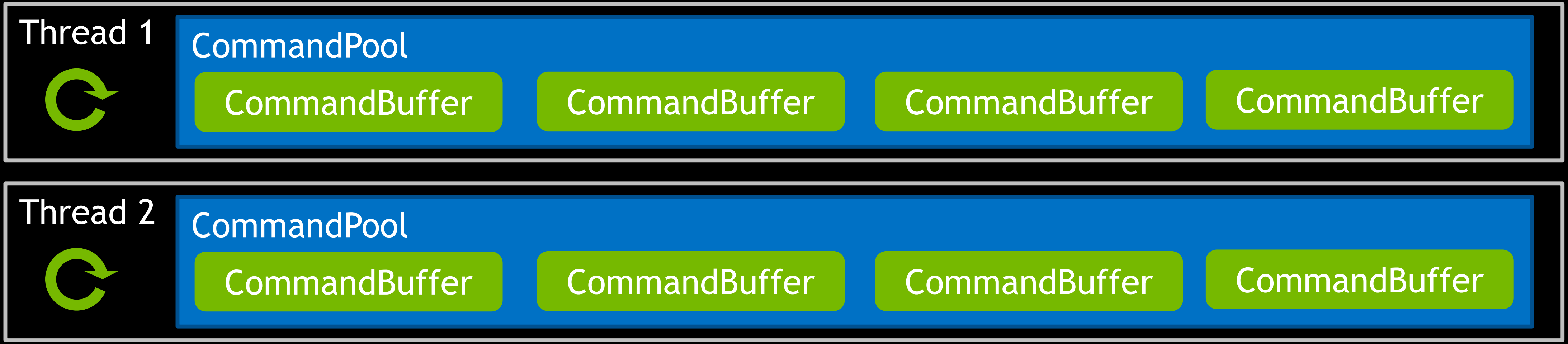
CommandBuffer

App Submissions to the Queue

Rewrite command buffer

Vulkan Threads: Command Pools

- VkCommandPool objects are pivotal to threaded command generation
- VkCommandBuffers are allocated from a “parent” VkCommandPool
- VkCommandBuffers written to in different threads must come from different pools
  - Or else both the buffer & pool must be externally synchronized, which isn’t worth it



Thread 1

CommandPool

CommandBuffer CommandBuffer CommandBuffer CommandBuffer

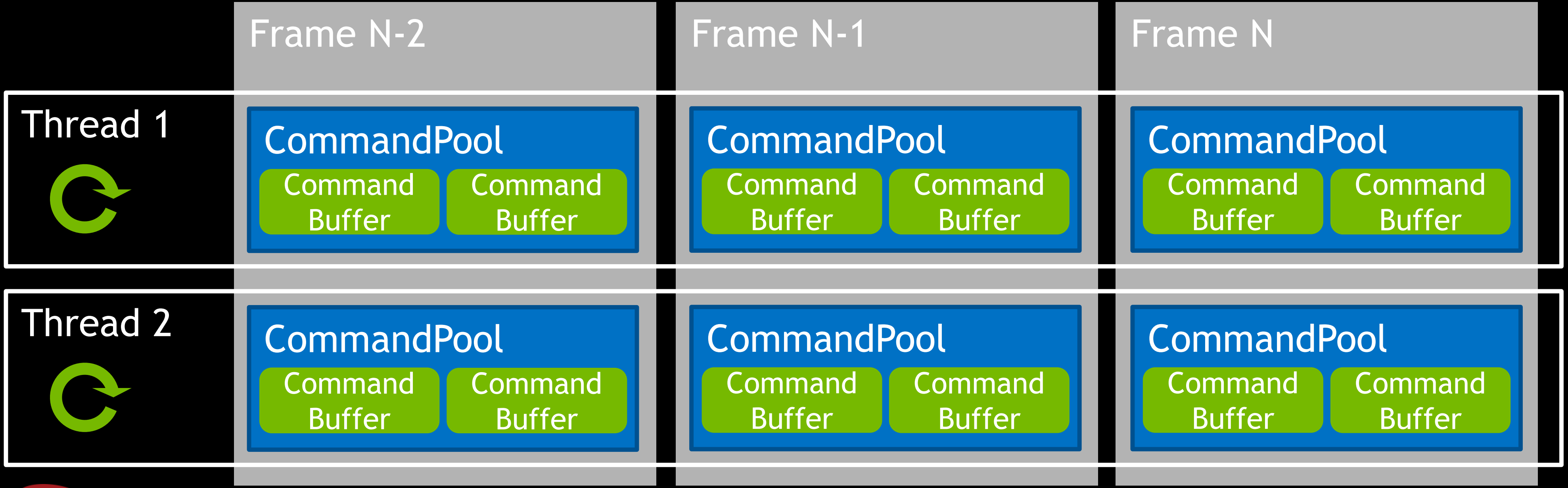
Thread 2

CommandPool

CommandBuffer CommandBuffer CommandBuffer CommandBuffer

Vulkan Threads: Command Pools

- Need to have multiple command buffers per thread
  - Cannot reuse a command buffer until it is no longer in flight
- And threads may have multiple, independent buffers per frame
- Faster to simply reset a pool when that thread/frame is no longer in flight:



Vulkan Threads: Descriptor Pools

- VkDescriptorPool objects may be needed for threaded object state generation
  - E.g. dynamically thread-generated rendered objects
- Pools can hold multiple types of VkDescriptorSet
  - E.g. sampler, uniform buffer, etc
  - Max number of each type specified at pool creation
- VkDescriptorSets are allocated from a “parent” VkDescriptorPool
  - descriptors allocated in different threads must come from different pools
- But VkDescriptorSets from the same pool can be written to by different threads

Vulkan Multi-Threading

QUESTIONS?