Vulkanised 2026

The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

Solving All Synchronisation Problems with Timeline Semaphores

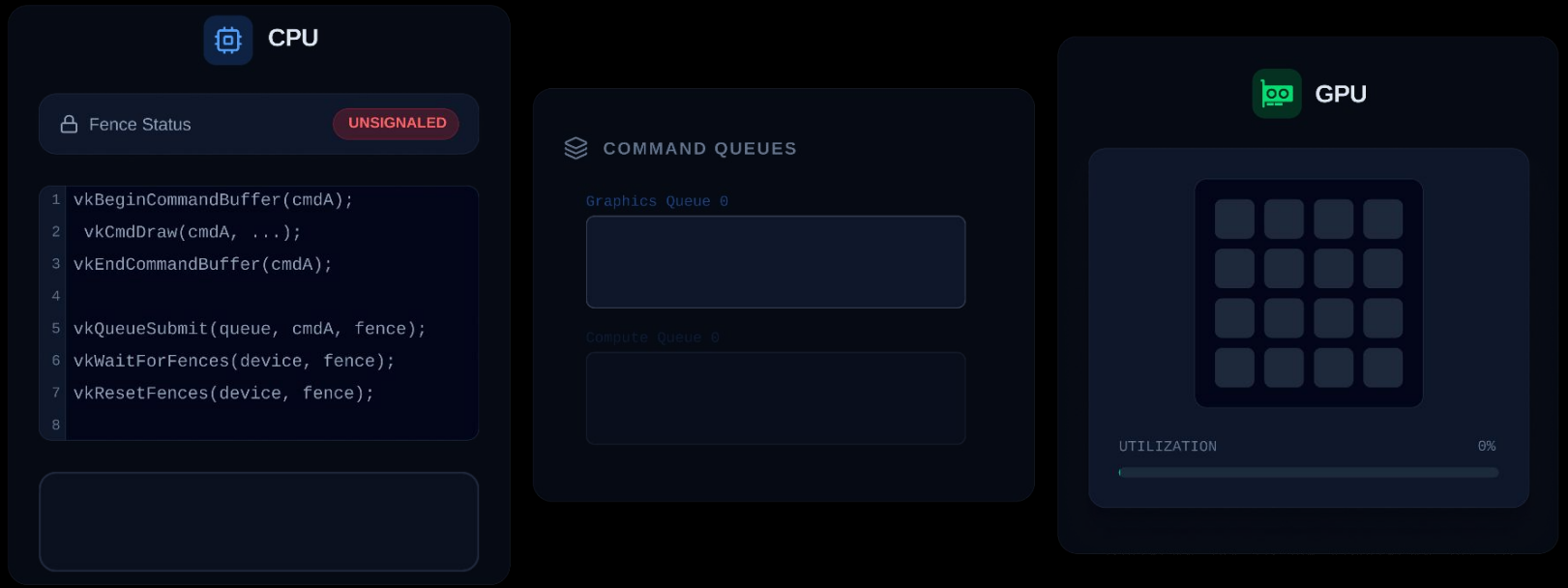
Lucas Miguel Antunes da Silva, Devsh Graphics Programming Sp. z O.O.

What is this talk about?

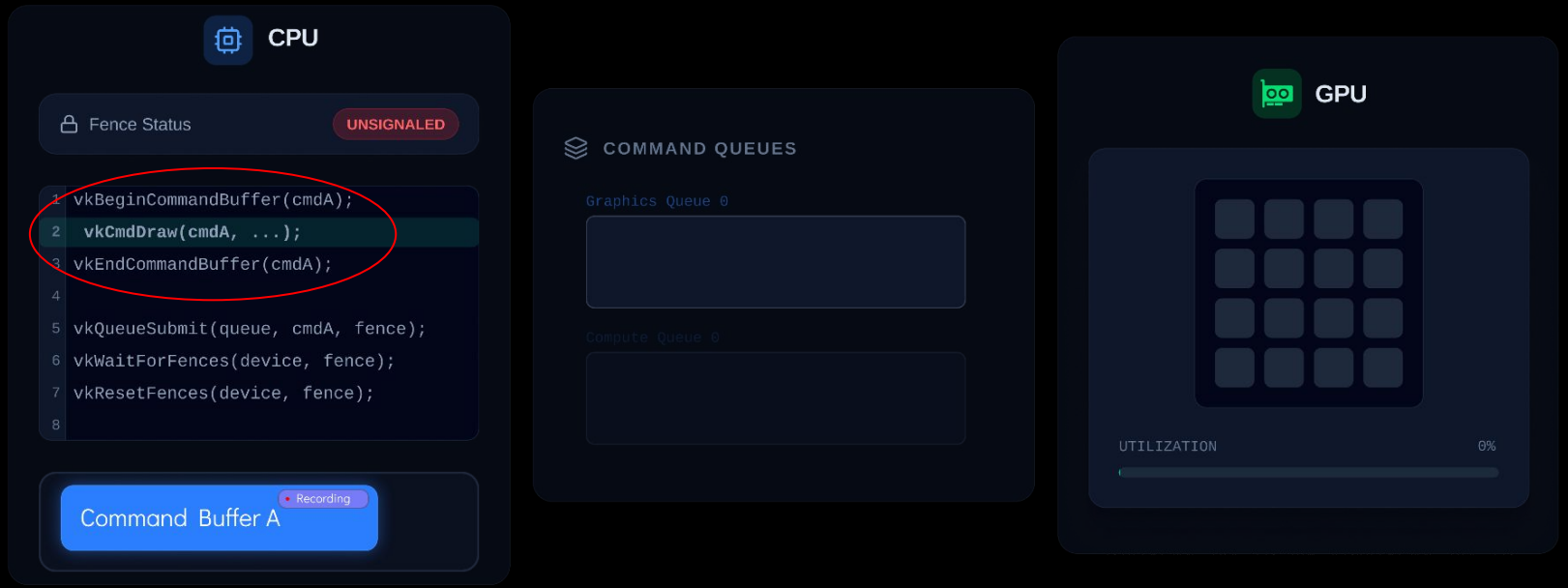
- Recap - the primitives
  - The limitations of **binary fences and semaphores** , and why they are **hard to scale** .
  - Why **timeline semaphores** exist and why you should start using them.

- Timeline semaphores as the backbone
  1. Connecting synchronization to **GPU resource lifetime and safe destruction**
  2. Practical, production-proven patterns from Nabla
     a. **Timeline Event Handlers:** Host callbacks, seemingly from device
     b. **"Intended" Submit** : automatic submission on overflow and how it enabled "immediate-mode" drawing for unpredictable data-sets
     c. **Safe Memory** or **Descriptor Set Allocations**

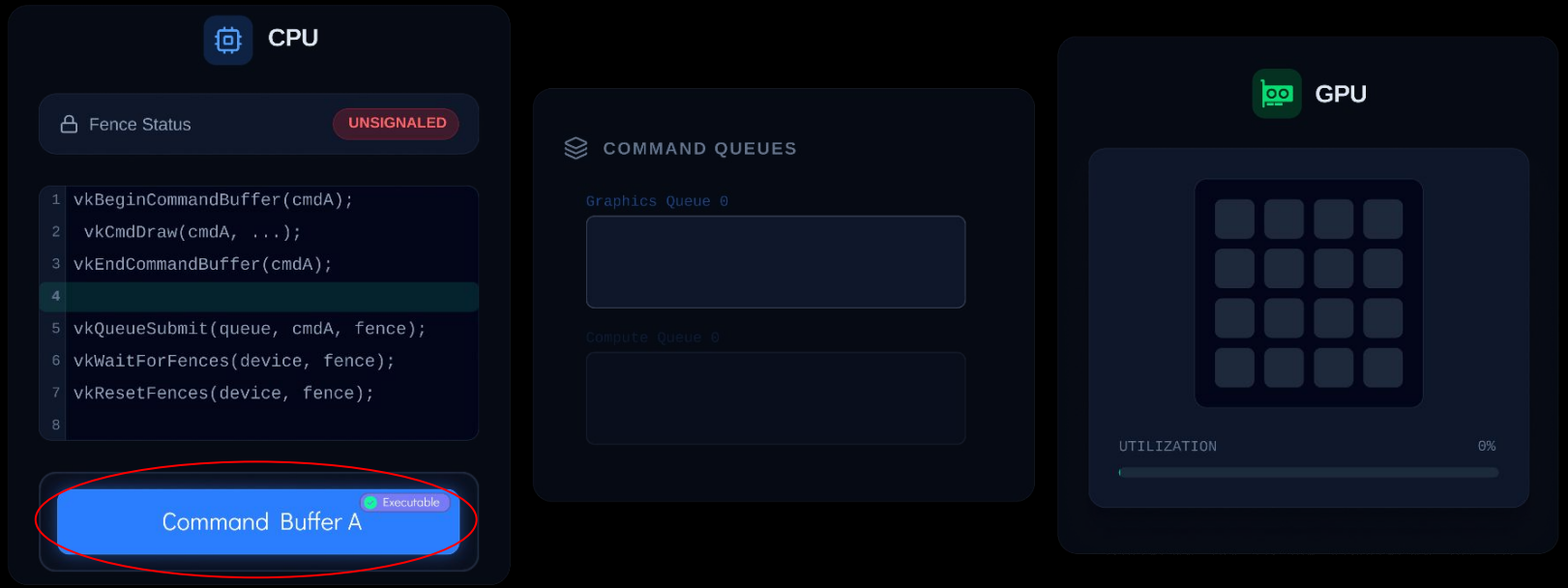
Simplified Mental Model🧠



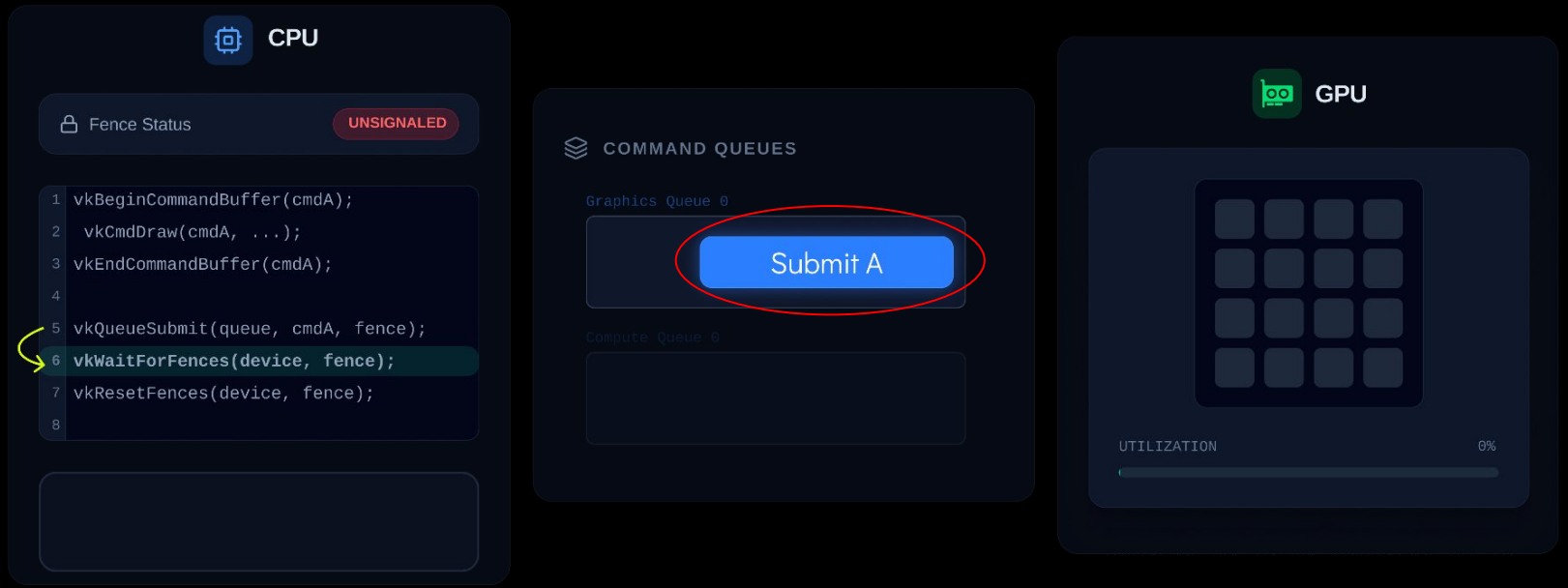
Recap: Fences



Recap: Fences

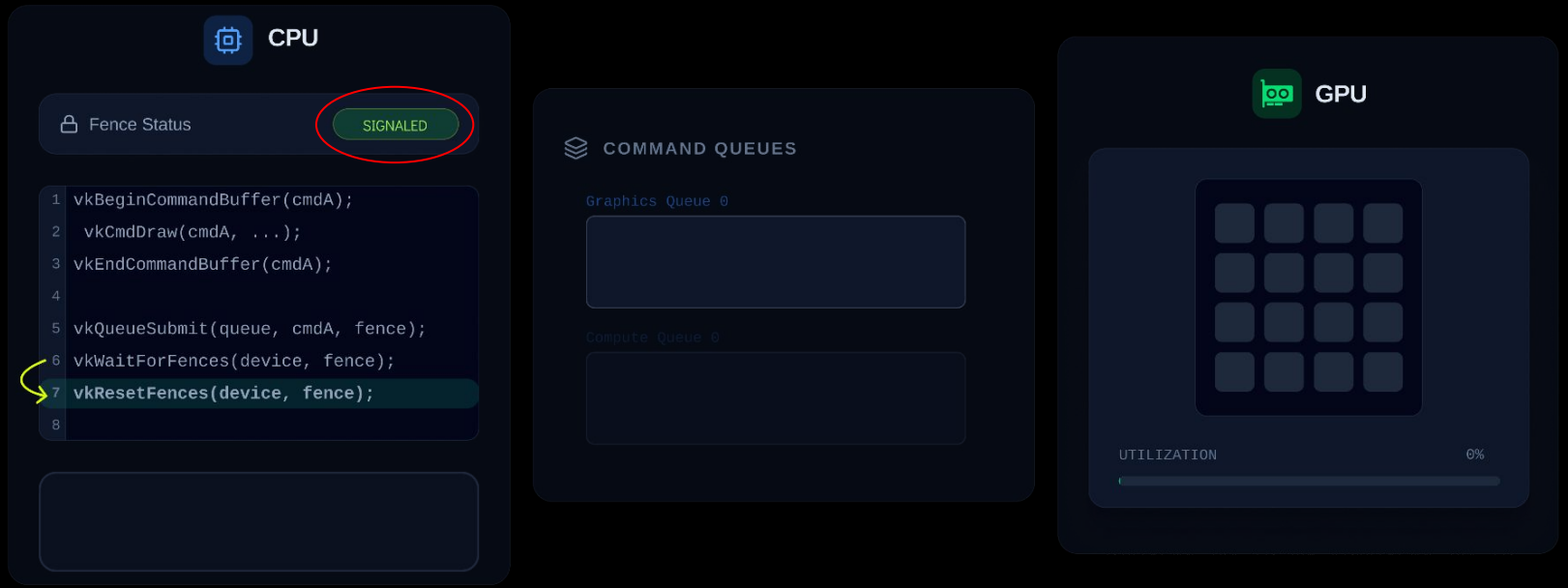


Recap: Fences

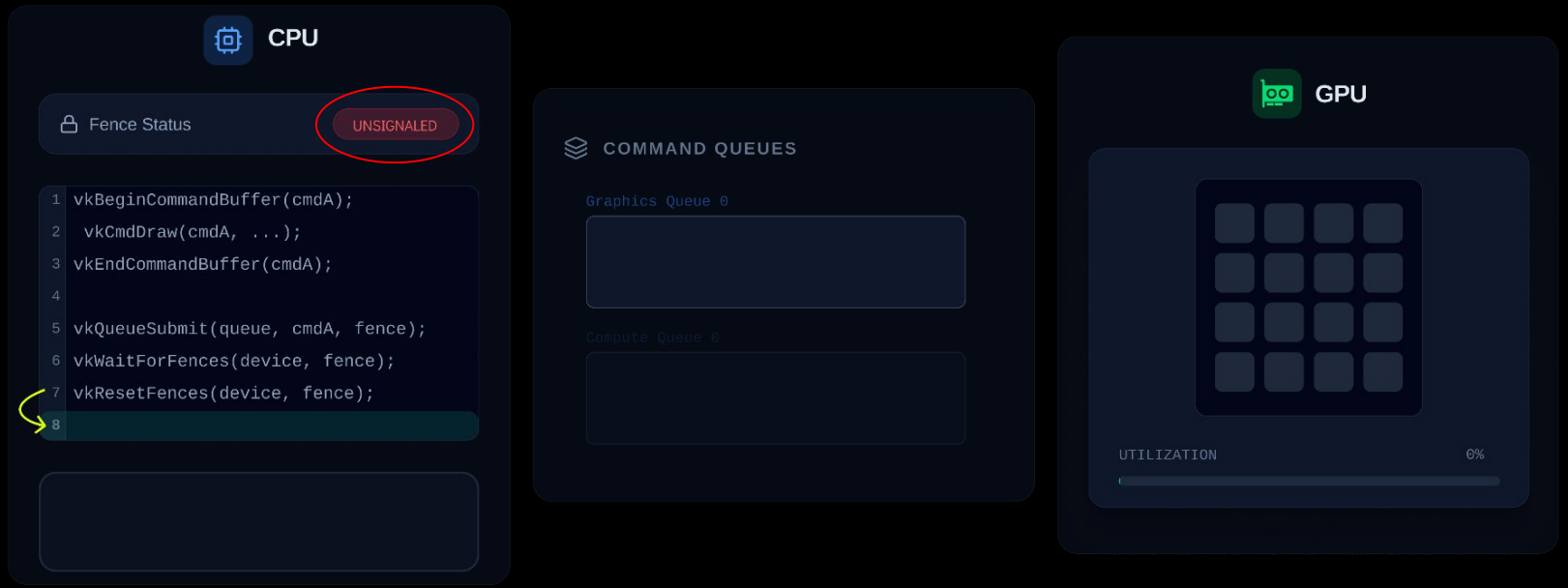


Recap: Fences

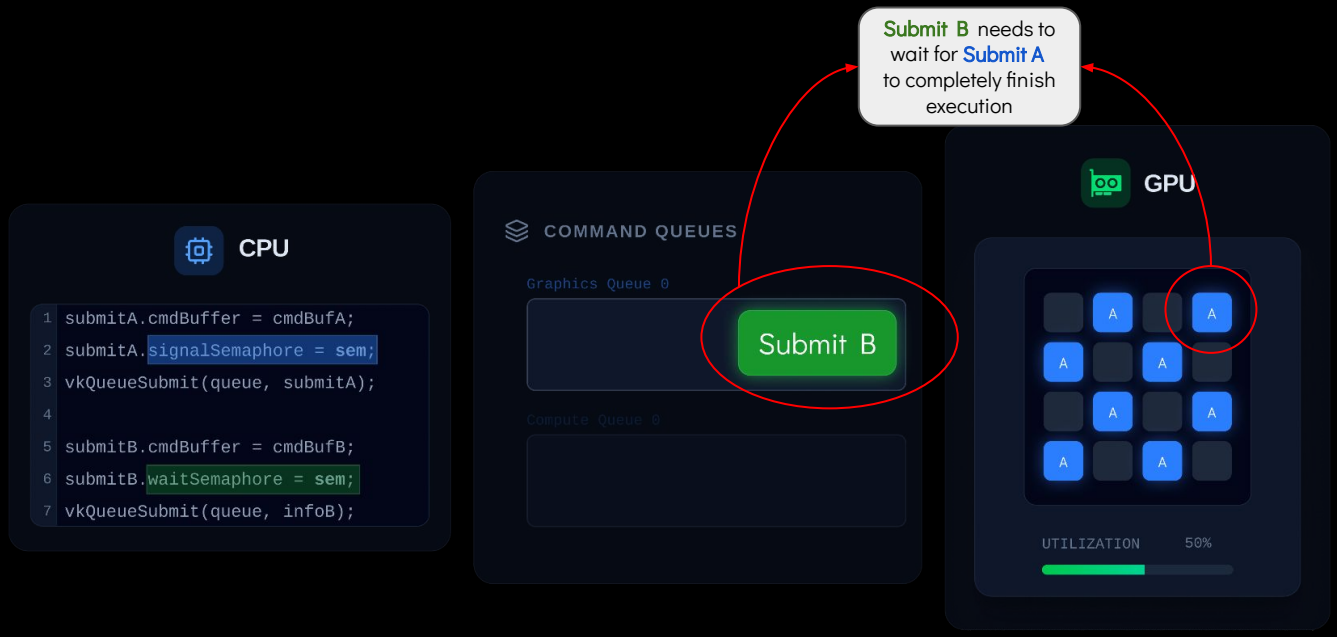
Recap: Fences



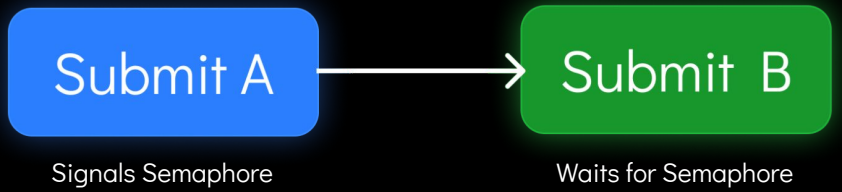
Recap: Fences



Recap: Binary Semaphores

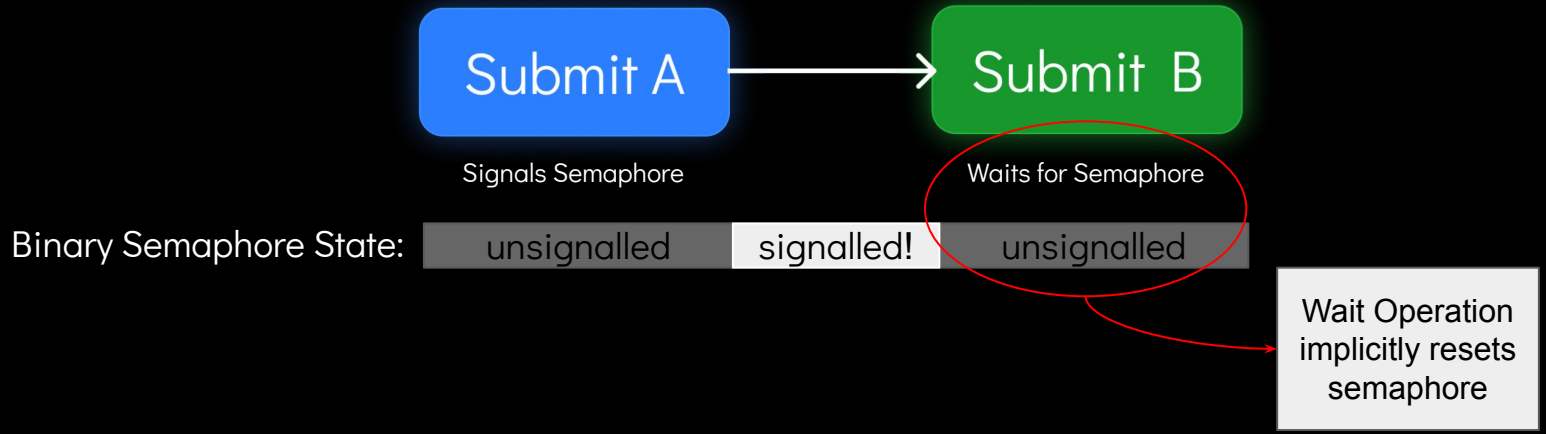


Recap: Binary Semaphores



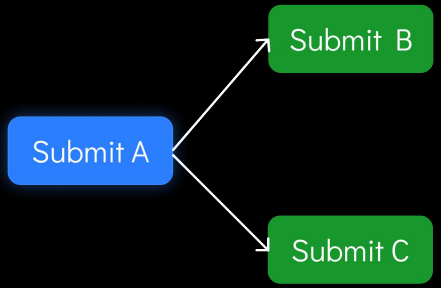
Recap: Binary Semaphores

Recap: Binary Semaphores



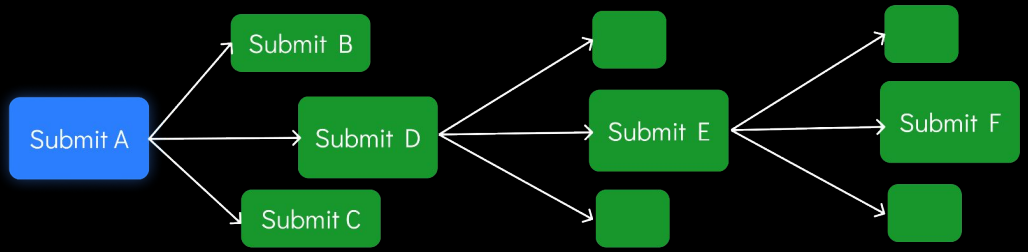
Recap: Limitations of Binary Semaphores

- We need **2 binary semaphores** here.
  - Reason: There needs to be 1:1 mapping between signals and waits.



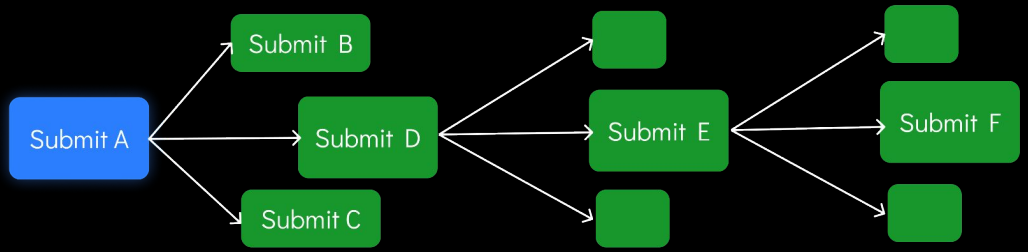
Recap: Limitations of Binary Semaphores

What about this? 🤣



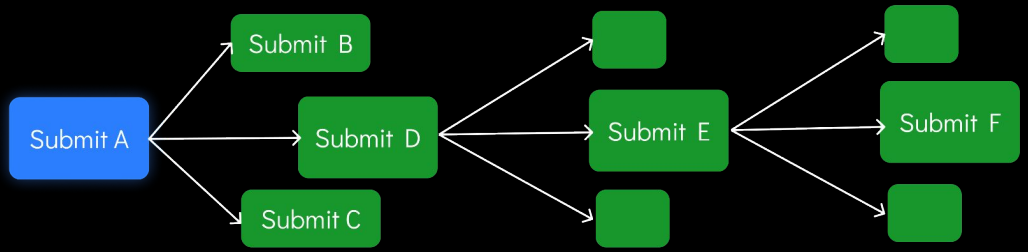
Recap: Limitations of Binary Semaphores

What about this? 🤣



Recap: Limitations of Binary Semaphores

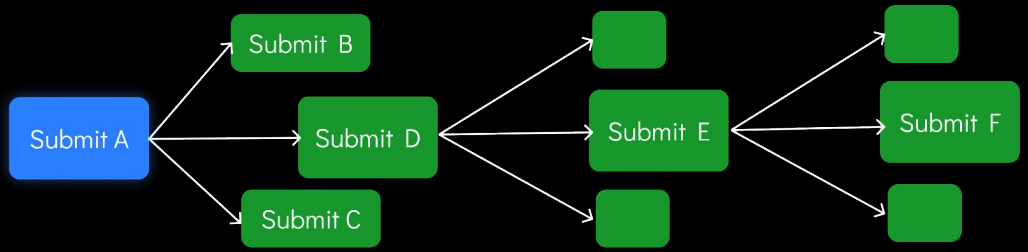
What about this? 🤣



- Number of Binary Semaphores needed:
  - Worst-case: a semaphore **per wait-signal pair** (9 in the figure)
  - Best-case: depends on how well you **recycle and reuse** old semaphores

Recap: Limitations of Binary Semaphores

What about this? 🤣

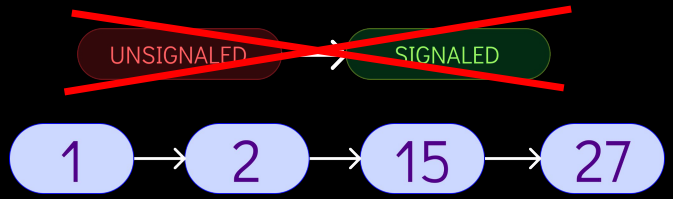


- Number of Binary Semaphores needed:
  - Worst-case: a semaphore **per wait-signal pair** (9 in the figure)
  - Best-case: depends on how well you **recycle and reuse** old semaphores

*If your submit dependencies are represented as a **Direct Acyclic Graph** , a single timeline semaphore can trace a subgraph in which vertices can have infinite **out-degree** but only **in-degree** of 1; Binary semaphore allows only out-degree of 1.*

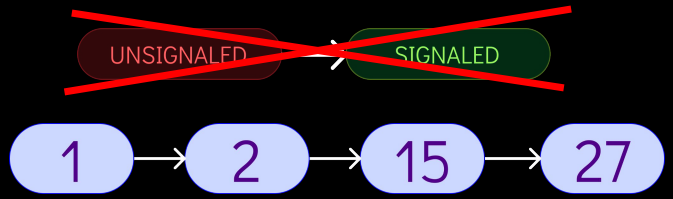
Recap: Timeline Semaphores to the rescue!

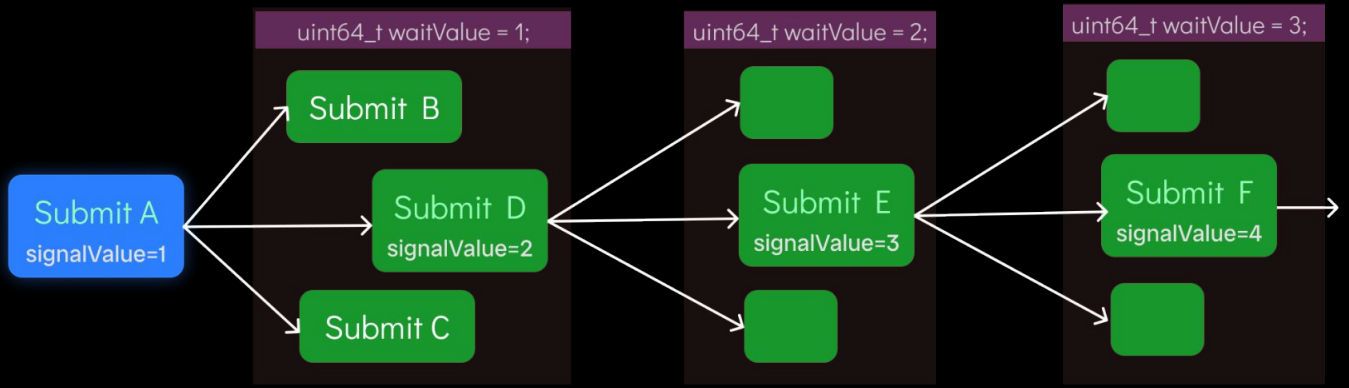
- Monotonically increasing uint64_t value.
- Signal a value, Wait on a value



Recap: Timeline Semaphores to the rescue!

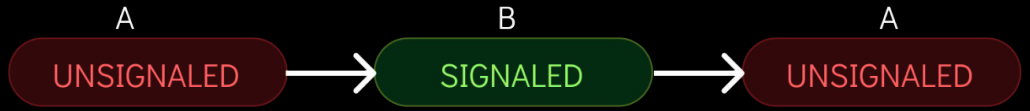
- Monotonically increasing uint64_t value.
- Signal a value, Wait on a value





Recap: Why should I care about Timeline Semaphores?

- Superset of Binary Semaphore and Fence (Can handle both CPU-GPU and GPU-GPU sync)
- One timeline semaphore per pine tree branch!
  - Plus, no need to worry about recycling your semaphores
- Possible to submit a wait for a value that **Host-signals after** (can't even signal binary semas from host!)
- No **ABA Signalling issue** with timeline semaphores



Fence is UNSIGNALED

Pending OR Post-Reset?

Recap: Why should I care about Timeline Semaphores?

Almost **every device** supports them!

(Devices that still get driver updates :D)

GPU Resource Management

Image, Buffer, Memory and Suballocations, Swapchain, Command Buffers, ...

Lifetime Tracking and Deletion at appropriate time.

GPU Resource Management

Skip this part if you know the **EXACT lifetime** of **ALL of your gpu objects**

Q: Why are we talking about **Resource Management** here?

Destroy resource that's ***in use by the GPU***

CRASH !

Destroy a resource ***already recorded into Command Buffer***

Invalid Command Buffer

We need Synchronization!

Deletion Queues

```
std::deque<callback_t> deletionQueue;
```

```
deletionQueue.push_back([=] { vkDestroyBuffer(...); });
```

1. Push deletion function to queue

```
// ... waiting on a timeline semaphore
vkWaitSemaphores(device, &waitInfo, ...);
```

2. Wait on a timeline semaphore to make sure GPU work has completed

```
// ... cleanup after GPU completion
while (!deletionQueue.empty()) {
    deletionQueue.front()();
    deletionQueue.pop_front();
}
```

3. Process the queue and call the callbacks (deletion functions).

Deletion Queues, a bit better

```
std::deque<std::pair<VkSemaphoreWaitInfo, callback_t>> dq;
```

WaitInfo + Callback Pair

```
dq.push_back({
    waitInfo(timelineSem, signalValue),
    [=]{ vkDestroyBuffer(...); }
});
```

1. Push waitInfo + callback to queue **(Latch)**

```
uint64_t cur = vkGetSemaphoreCounterValue(device, timelineSem);
while (!dq.empty() && dq.front().first.value <= cur) {
    dq.front().second();
    dq.pop_front();
}
```

2. Pop callback values <= cur **(Poll)**

Deletion Queues

Limitations:

- User still needs to **manually** queue for deletion

```
deletionQueue.push_back([=] { vkDestroyBuffer(...); });
```

- Objects reference each other
  - **Freeing a memory** that has buffers still bound to it?
  - **Destroying an Image** that still has ImageViews?
  - **Destroying a Buffer** that still has BufferViews?
  - Need to manually write cleanup function that **DEALS WITH EVERYTHING you reference**

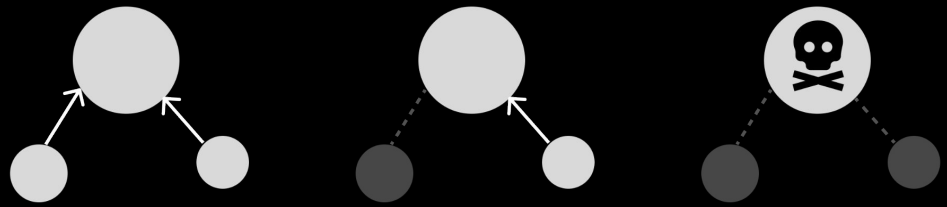
Automated Lifetime Tracking

Reference Counting

- The "Cycle" problem is a non-issue: Vulkan/GPU object hierarchies are Directed Acyclic Graphs

~~Tracing Garbage Collection (Mark+Sweep)~~

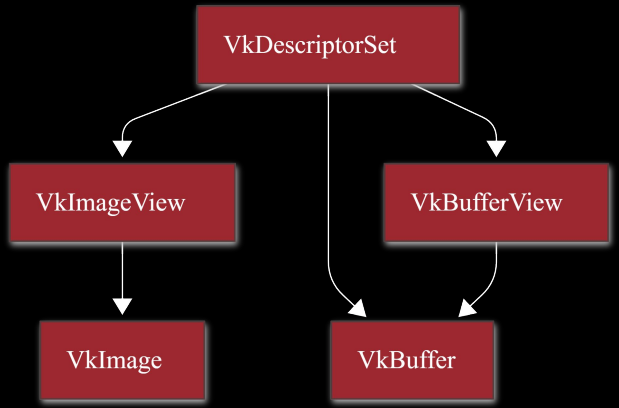
- ✖ “Stop-the-world” stutters
- ✖ Non-deterministic
  - (e.g delete VkDevice before VkBuffer)



Automated Lifetime Tracking

Reference Counting

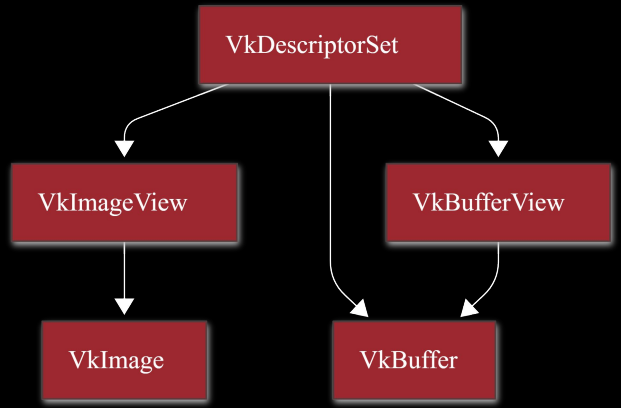
- The "Cycle" problem is a non-issue: Vulkan/GPU object hierarchies are Directed Acyclic Graphs



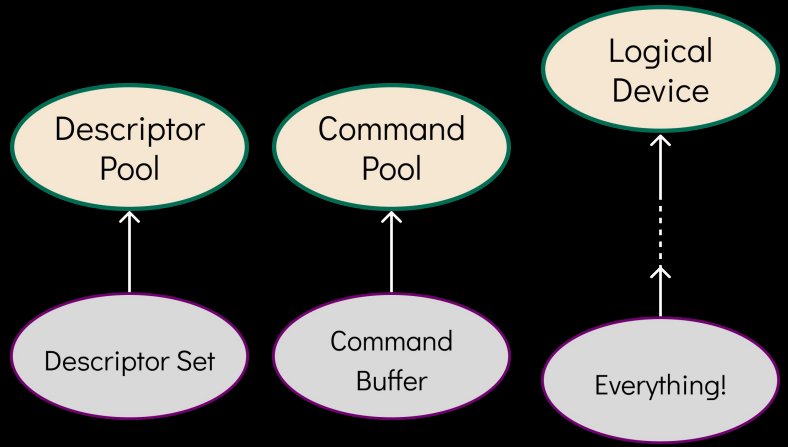
Automated Lifetime Tracking

Reference Counting

- The "Cycle" problem is a non-issue: Vulkan/GPU object hierarchies are Directed Acyclic Graphs



+ Backward
(keep factory alive)



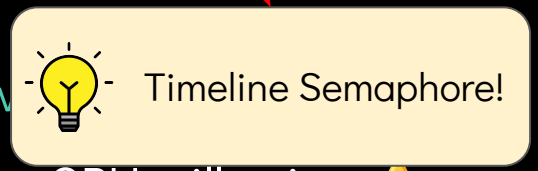
CPU Lifetimes ≠ GPU Lifetime

Reference Counting handles the object's lifetime on CPU.
but **it is blind to the GPU's timeline**

- **Most common pitfall** with RAII (e.g. vulkan hpp's vk::raii)
  - vk::raii::Image out-of-scope, gets destroyed but GPU still using ⚠️
  - vk::raii objects don't even reference count other objects on CPU.
    - *If Image* goes out of scope -> *ImageView* becomes Invalid

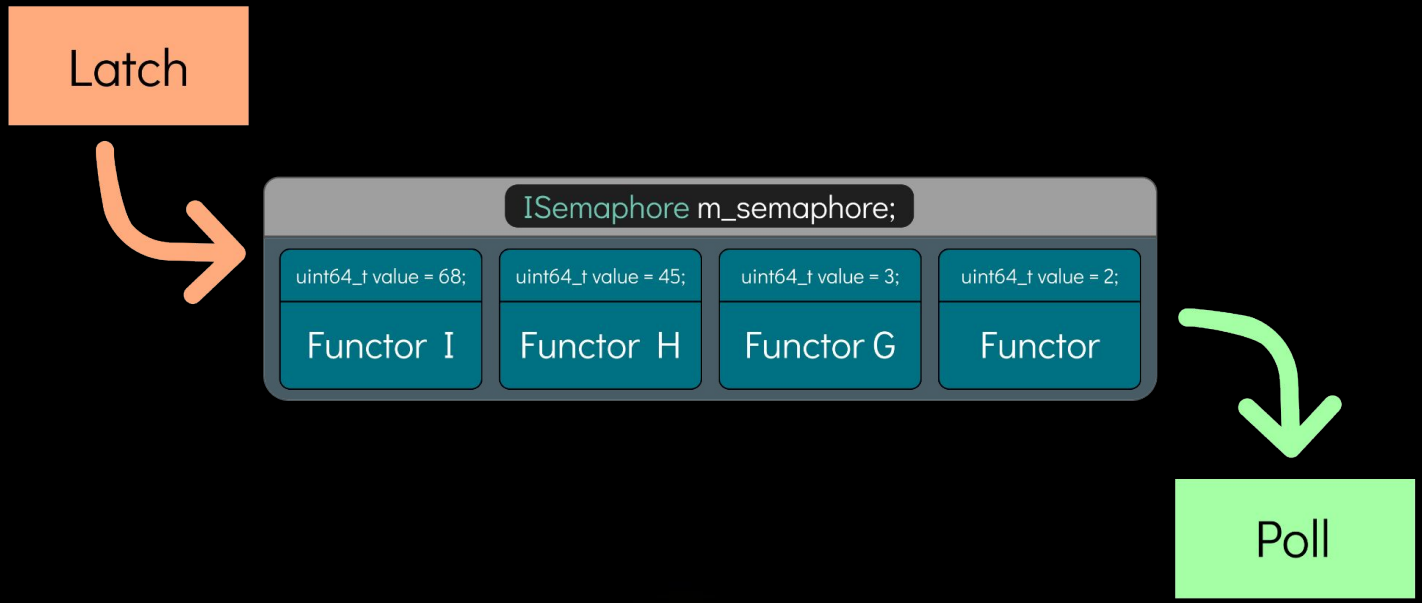
CPU Lifetimes ≠ GPU Lifetime

Reference Counting handles the object's lifetime on CPU.
but **it is blind to the GPU's timeline**

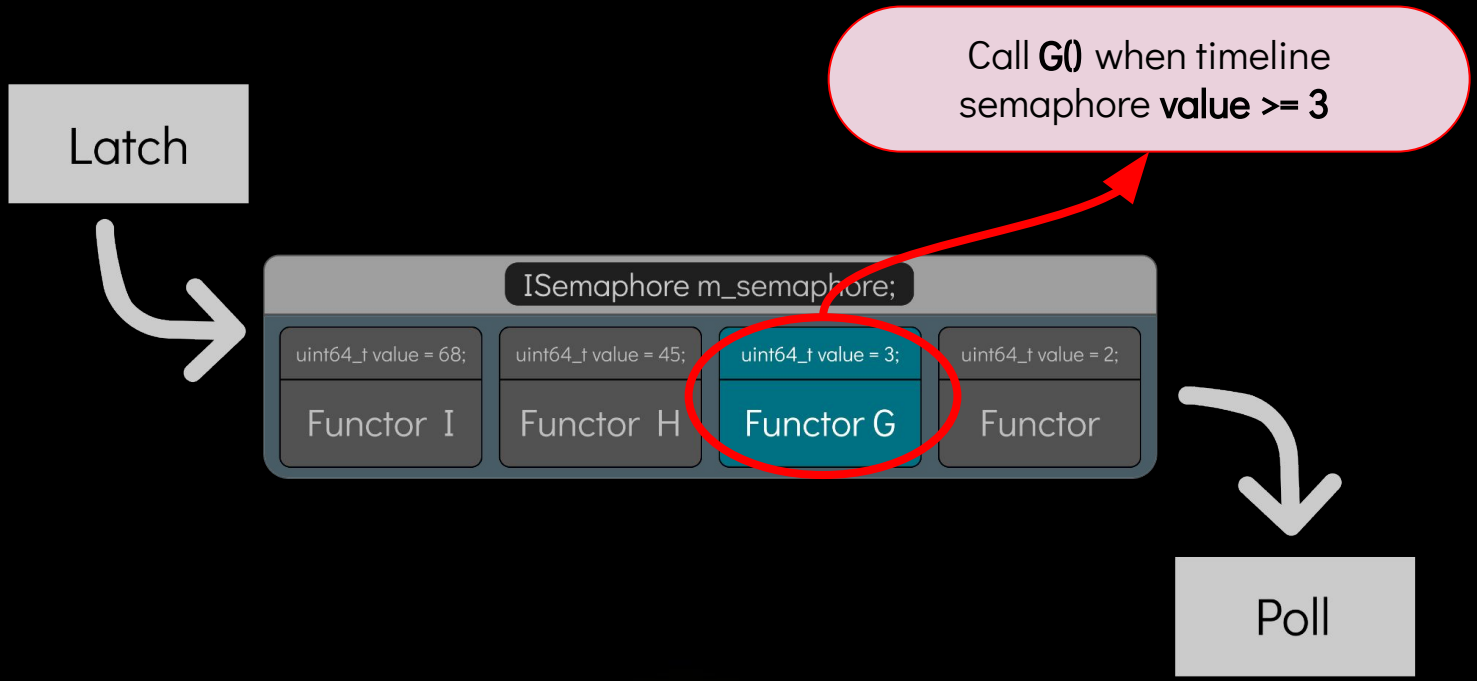


- **Most common pitfall** with RAII (e.g. vulkan hpp's v
  - vk::raii::Image out-of-scope, gets destroyed but GPU still using ⚠️
  - vk::raii objects don't even reference count other objects on CPU.
    - *If Image* goes out of scope -> *ImageView* becomes Invalid

The Solution: Timeline Event Handler



The Solution: Timeline Event Handler

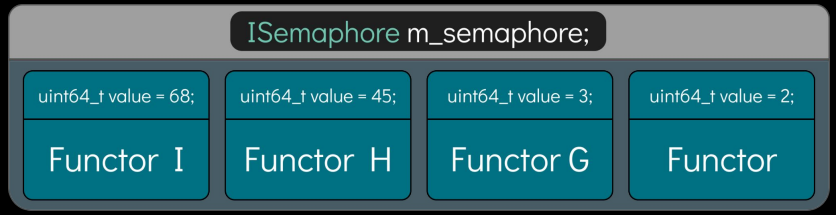


The Solution: Timeline Event Handler

`template<typename Functor>` `class TimelineEventHandler`	
`void latch(uint64_t, Functor)`	Enqueue a `semaValue` + `Functor` pair. 
`PollResult poll()`	Processes the queue. Pops and executes **functors** with **`Value <= m_greatestSignalValue`**
`void wait(time_point timeout_time)`	It provides **incremental** , **time-bounded** semaphore waits and callback calls to avoid "stop-the-world" stutters. (assumes callbacks takes equal time)

The Solution: Timeline Event Handler

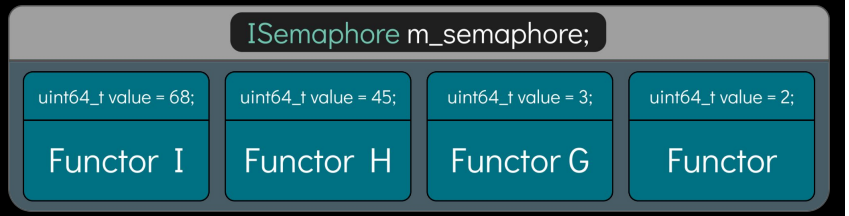
Single TimelineEventHandler



Multi TimelineEventHandler

The Solution: Timeline Event Handler

Single TimelineEventHandler

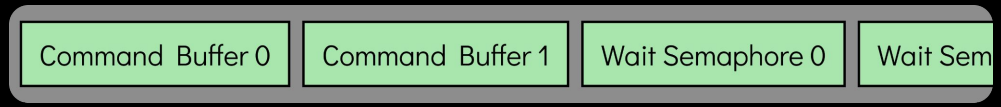


Multi TimelineEventHandler

TimelineEventHandler to track resources on GPU timeline

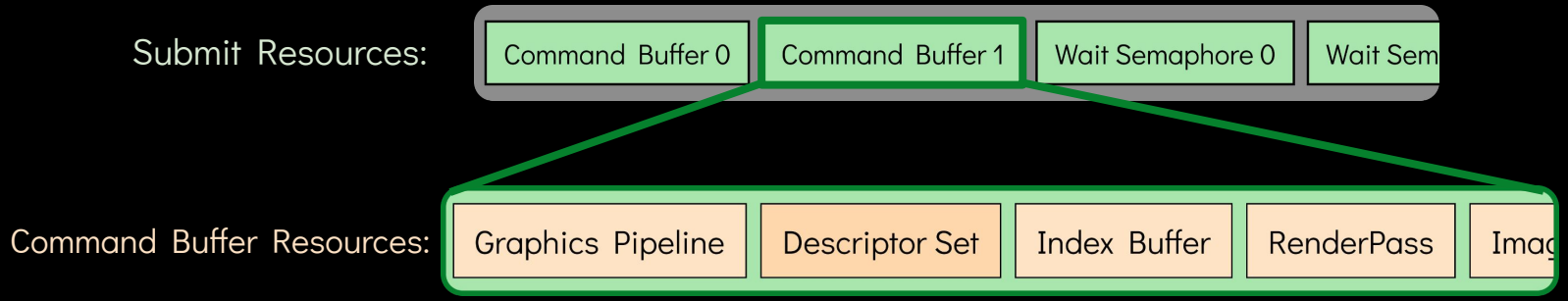
- Queue(VkQueue Wrapper) has a **built-in TimelineEventHandler**
  - Queue Latches submit resources + signal semaphore.
  - Queue Polls after each submit

Submit Resources:



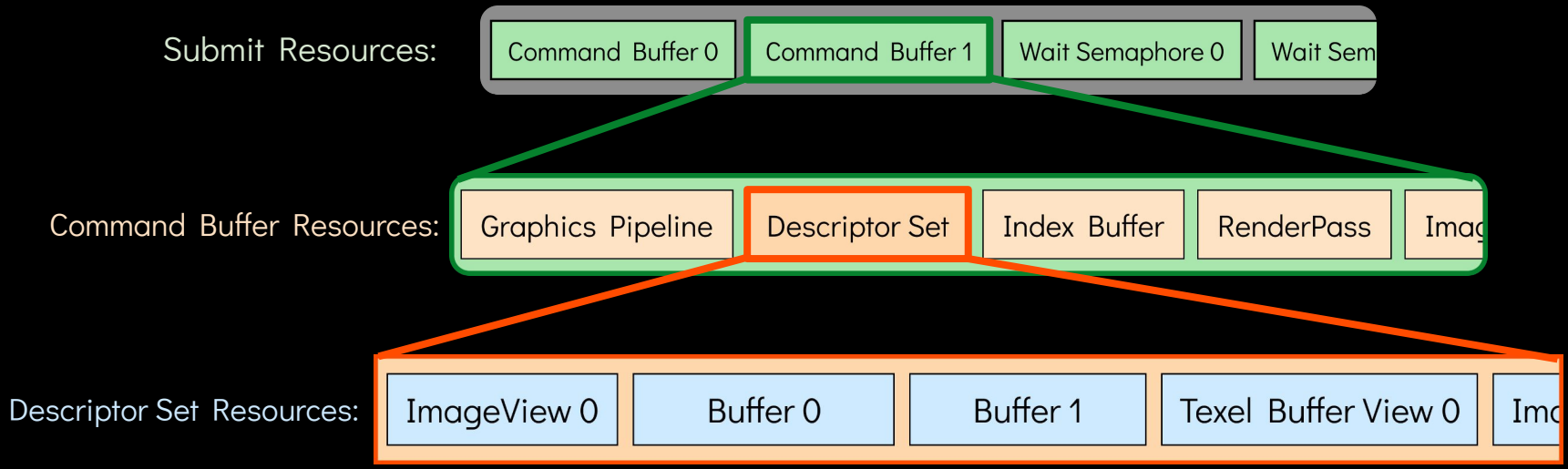
TimelineEventHandler to track resources on GPU timeline

- Queue has a **built-in TimelineEventHandler**
  - Queue Latches submit resources + signal semaphore.
  - Queue Polls after each submit



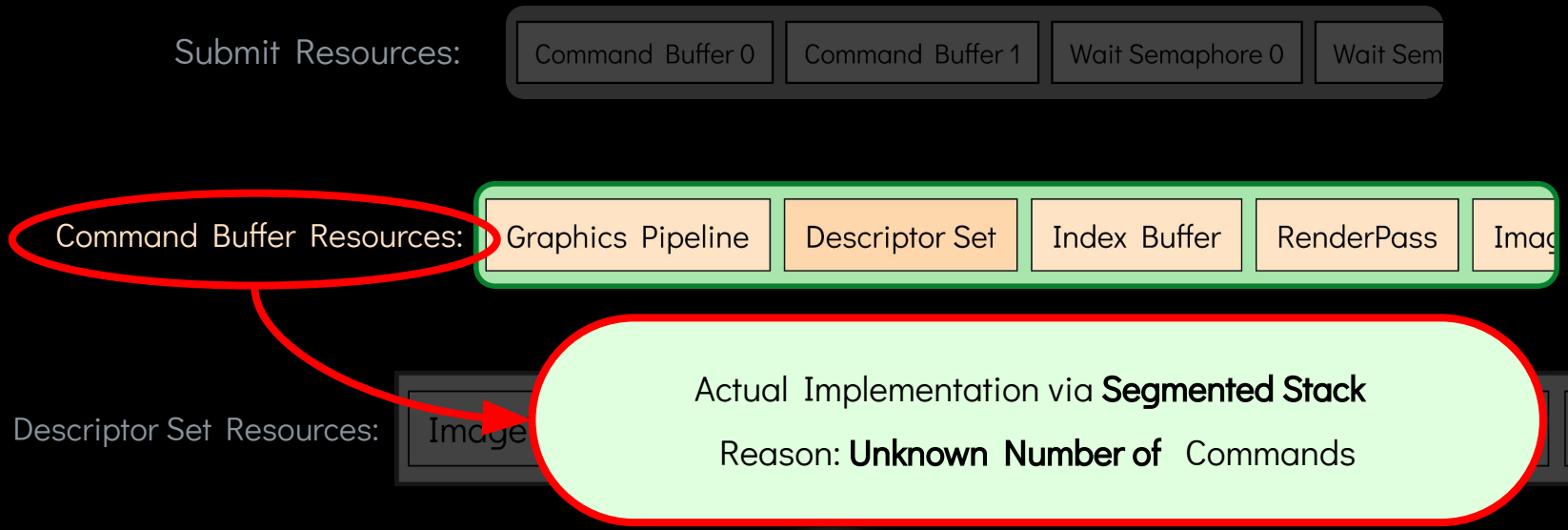
TimelineEventHandler to track resources on GPU timeline

- Queue has a **built-in TimelineEventHandler**
  - Queue Latches submit resources + signal semaphore.
  - Queue Polls after each submit



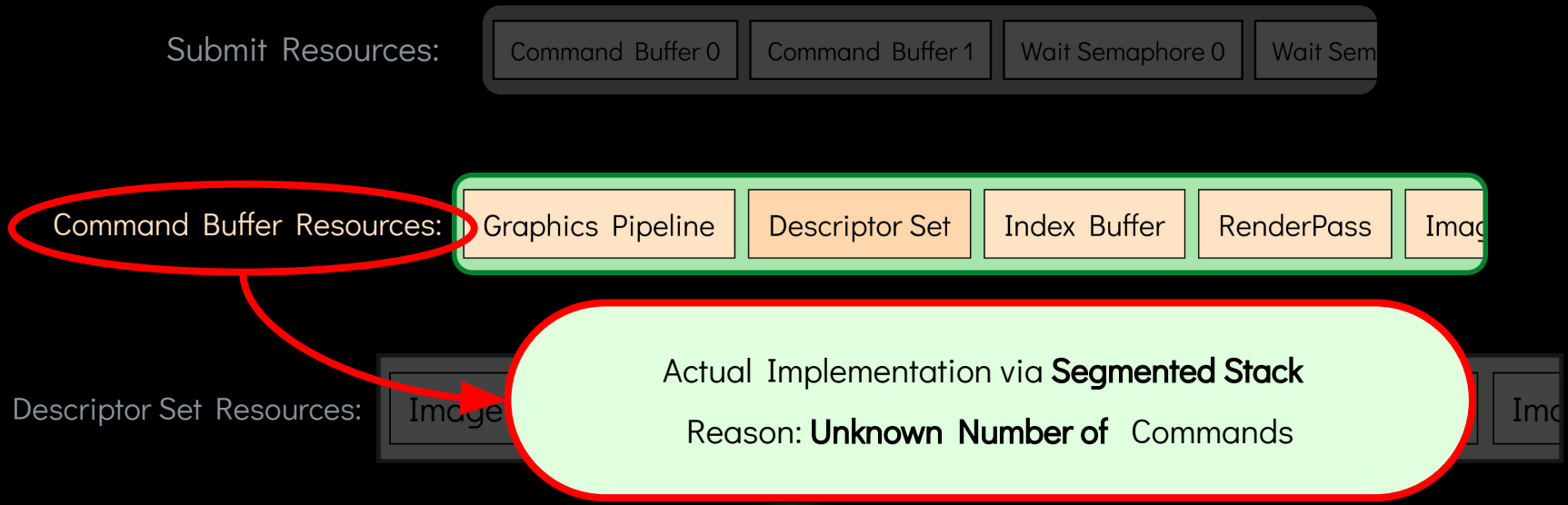
TimelineEventHandler to track resources on GPU timeline

- Queue has a **built-in TimelineEventHandler**
  - Queue Latches submit resources + signal semaphore.
  - Queue Polls after each submit



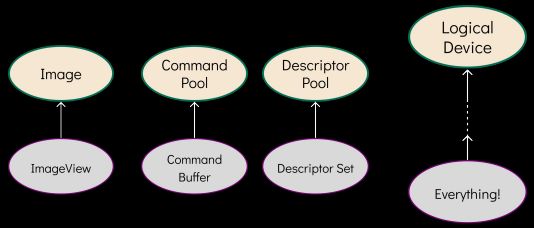
TimelineEventHandler to track resources on GPU timeline

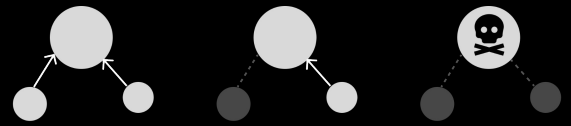
- Lifetime tracking needs special checks for Command Buffer invalidation
  - Compare atomic reset counter of Parent Pool vs last known at vkCmdBeginCommandBuffer
  - A record of bound descriptor sets with any non UPDATE_AFTER_BIND bindings

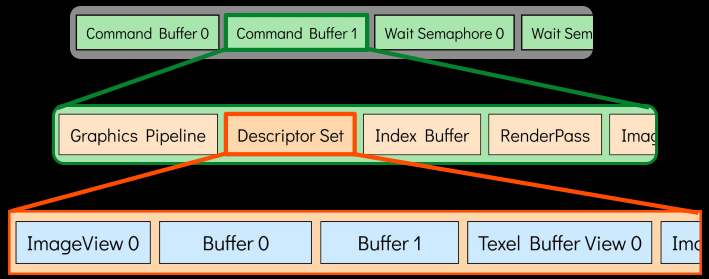


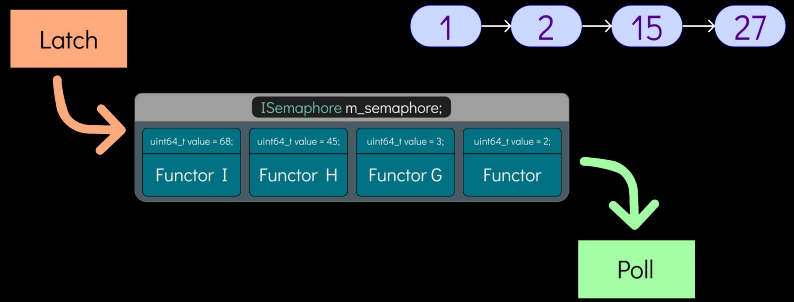
TL;DR

It's similar to **shared_ptr** but the lifetime is also determined through **timeline semaphore** .
(with help of TimelineEventHandler)









Stutters or Deletion Stop-the-world Cascades on Poll

Avoid Stutters and Deletion Cascades on Poll

1. You can cancel or **bail** on **Poll** midway
   - **Functor** returns true → Cancel polling
   - Or exceeded the time limit
   - Example: enough memory was freed, no need to poll any further

1. **Bucketing** into **separate** TimelineEventHandlers.
   - Example of **two buckets** for different purposes:

Limitations and Future work

1. Reference Leaking
   - Careful to **nullify descriptors** in a set you're not going to use anymore
   - *Future work:* Debug Tools for detecting leaks.

1. Race Condition with multi-threaded submits
   - Polling in multiple threads **may touch the same resource** if objects share a parent/manager at destruction. (eg. different descriptor sets, same descriptor pool)
   - *Future work:* Synchronize access to parent pool/manager in destructors

1. Cycles are still possible!
   - LogicalDevice → Queue → CommandBuffer → LogicalDevice
   - Solution: Queues need to be polled to exhaustion before LogicalDevice goes out of scope

More Timeline Event Handlers!

Back to the Future: Fire & Forget

- Implemented with **TimelineEventHandler**

```
ISemaphore::future_t<GPUBufferHandle> create_and_initialize_buffer(...)
{
    // SETUP STUFF
    //...
    auto signalSemaphore = device->createSemaphore(0);
    //...
    //...
    cmdBuffer->updateBuffer(buffer.get(), 0, size, data);
    cmdBuffer->end();
    transferQueue->submit({ &cmdInfo, signalSemaphore });

    return ISemaphore::future_t<GPUBufferHandle>(signalSemaphore, buffer);
}
```

1. Setup: Create buffer, allocate + bind memory, and setup command pool/buffer

*2. Create temporary timeline semaphore**

3. Record & Submit

4. Construct future_t instead of blocking for semaphore

Back to the Future: Fire & Forget

future_t wouldn't work with binary fences!

Fence is UNSIGNALED

Pending OR Post-Reset?

* *It can work when fences are never reset OR destroyed (vkFence handles are unique)*

CUDA stream host callbacks

- Implemented via **Queue**'s built-in **Timeline Event Handler**
  - CUDA code example:

```
Stream s1;
```

⇔ a single "timeline"

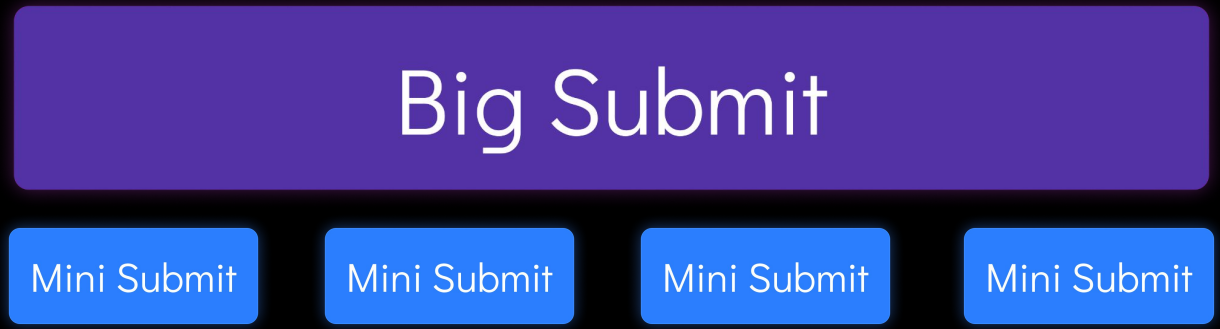
```
// push kernel to queue/stream and continue
gpu_kernel<<<grid, block, s1>>>(data);
```

⇔ vulkan Submit of compute work

```
// Call "MyHostFunction" when gpu
// is done executing the kernel
add_host_callback(s1, MyHostFunction);
```

⇔ **Latching** a callback to **TimelineEventHandler** with the previous submit's signal semaphore!

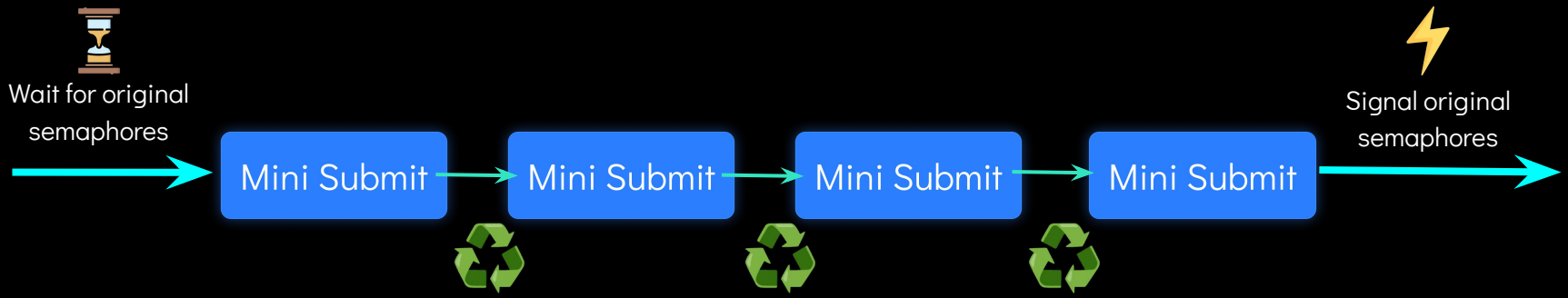
Breaking Up with Big Submits



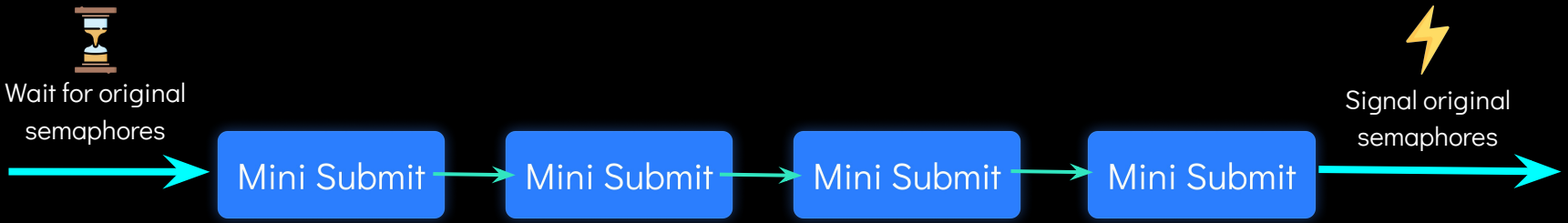
- Why? → Limited Resources
  - *Example:* Not enough VRAM to render the full complex scene
  - *Example:* Squeezing a **1 GB** texture through a small staging buffer.

Breaking Up with Big Submits

Breaking Up with Big Submits



Breaking Up with Big Submits



Breaking Up with Big Submits

What command buffers to use for our Mini Submits ?

Mini Submit | Mini Submit | Mini Submit | Mini Submit

Breaking Up with Big Submits

What command buffers to use for our Mini Submits ?

Mini Submit | Mini Submit | Mini Submit | Mini Submit

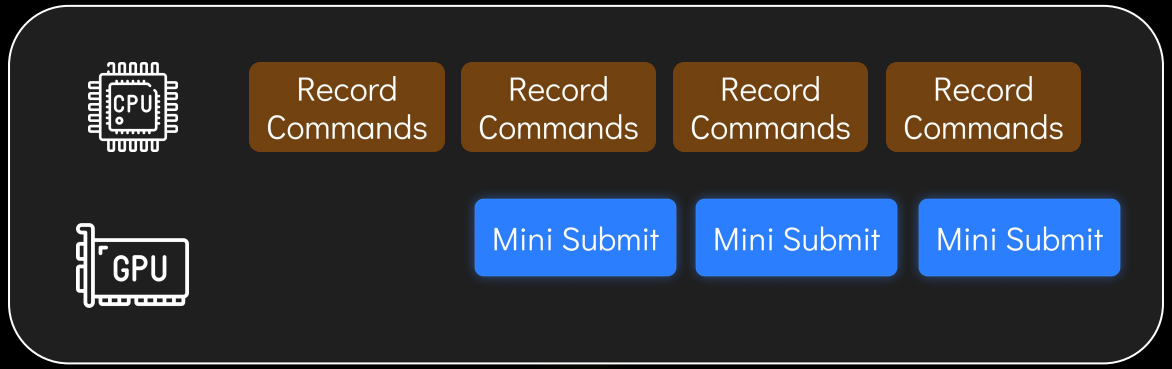
Reuse the last command buffer? 😢

Breaking Up with Big Submits

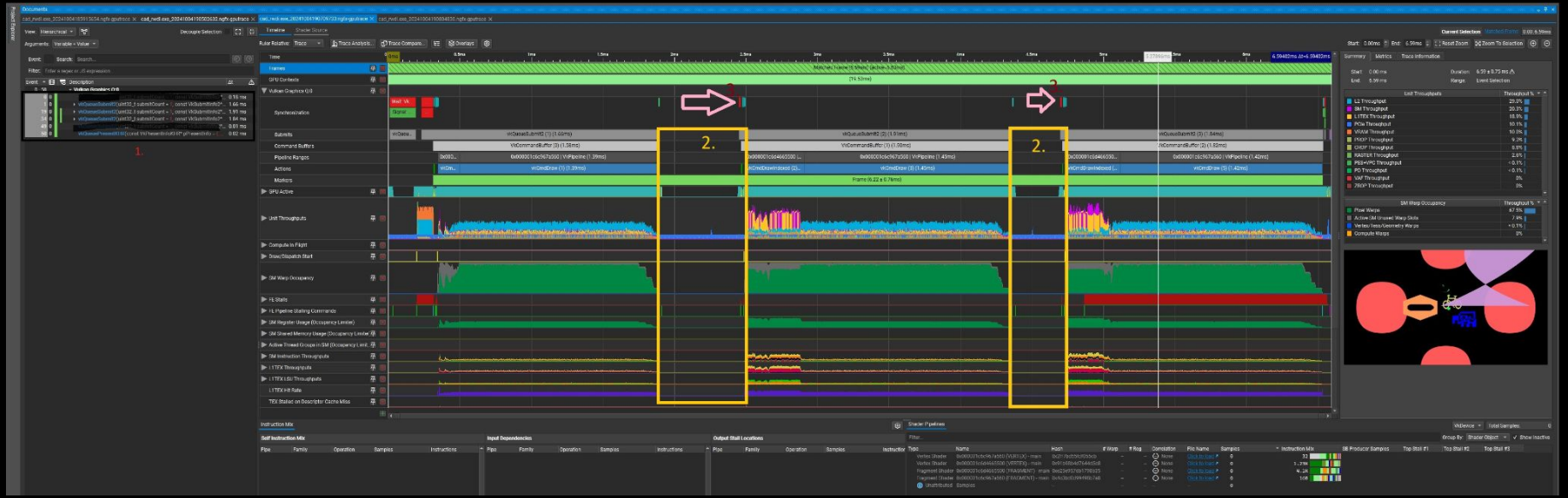
What command buffers to use for our Mini Submits ?

Mini Submit | Mini Submit | Mini Submit | Mini Submit

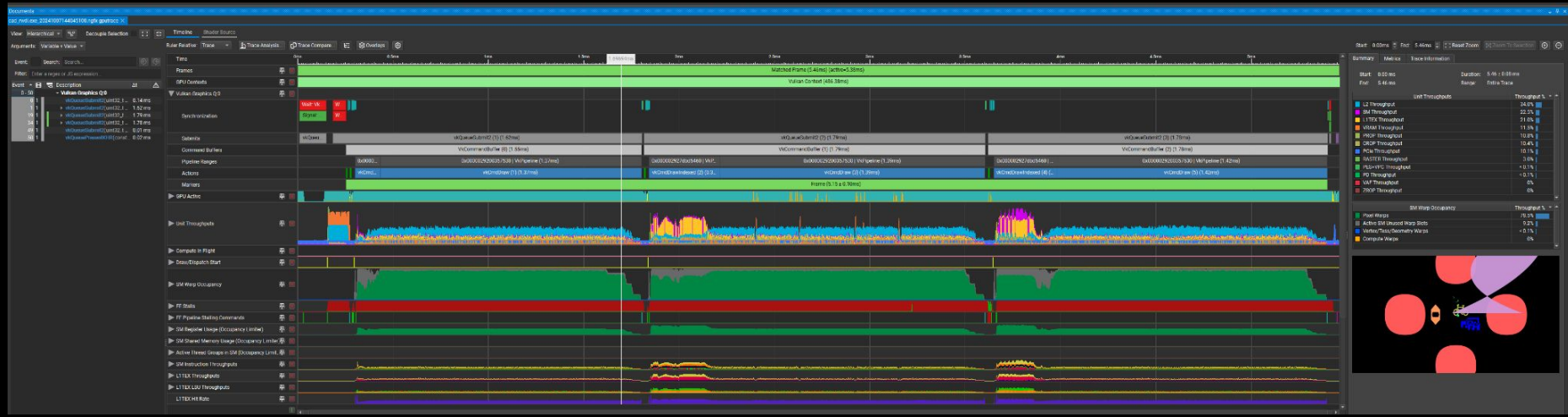
Round-Robin Multiple Command Buffers 💡😊



Breaking Up with Big Submits



Breaking Up with Big Submits



SIntendedSubmitInfo = Immediate Mode Rendering !

- SIntendedSubmitInfo
  - An wrapper around Vulkan's VkSubmitInfo
  - Logic for scratch semaphore
  - Logic to round-robin multiple command buffers
  - Function to **auto-submit** and start recording the next one immediately

- Limitations
  - Can't end command buffer in the middle of the renderpass
    - Forced to end renderpass and begin a new one on next submit
      - *Requires us to have 3 versions of a single renderpass (Begin, Continue, End)*
    - Dynamic Rendering Extension will fix this!

Ray Tracing Acceleration Structure Lifetime Tracking

- For TLAS we prefer to put them in Descriptor Sets
  - Already works with our existing Descriptor lifetime tracking

Ray Tracing Acceleration Structure Lifetime Tracking

- For TLAS we prefer to put them in Descriptor Sets
  - Already works with our existing Descriptor lifetime tracking

- Nabla's Command Buffer abstraction allows "recording" of arbitrary Shared Pointers which will be kept until Invalidation or Reset
  - This is how Buffers only referenced via Buffer Device Address (BDA) are kept alive
    - Need to remember to record, but can package all references in one object
  - You can keep a "pass by address" TLAS alive in the same way

Ray Tracing Acceleration Structure Lifetime Tracking

- For TLAS we prefer to put them in Descriptor Sets
  - Already works with our existing Descriptor lifetime tracking

- Nabla's Command Buffer abstraction allows "recording" of arbitrary Shared Pointers which will be kept until Invalidation or Reset
  - This is how Buffers only referenced via Buffer Device Address (BDA) are kept alive
    - Need to remember to record, but can package all references in one object
  - You can keep a "pass by address" TLAS alive in the same way

- But what about Bottom Level Acceleration Structures?

Ray Tracing Acceleration Structure Lifetime Tracking

- For TLAS we prefer to put them in Descriptor Sets
  - Already works with our existing Descriptor lifetime tracking

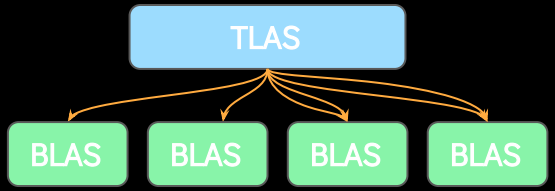
- Nabla's Command Buffer abstraction allows "recording" of arbitrary Shared Pointers which will be kept until Invalidation or Reset
  - This is how Buffers only referenced via Buffer Device Address (BDA) are kept alive
    - Need to remember to record, but can package all references in one object
  - You can keep a "pass by address" TLAS alive in the same way

- But what about Bottom Level Acceleration Structures?
  - References to them "leak" into the TLAS past the Submission Boundary
  - Device Build or Copy arguments can be in non-HOST_VISIBLE memory!

- Poses similar challenges to extending our Lifetime Tracking as EXT_descriptor_heap

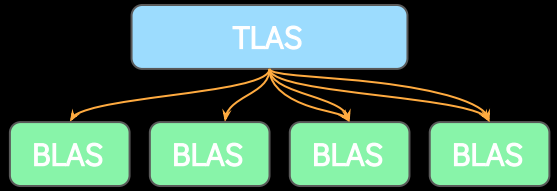
BLAS Lifetime Tracking by TLAS Instances

- Use set<shared_ptr<BLAS>> throughout
  - TLAS can instance a BLAS dozens of times
  - Avoids atomic operation spam by ref. counting

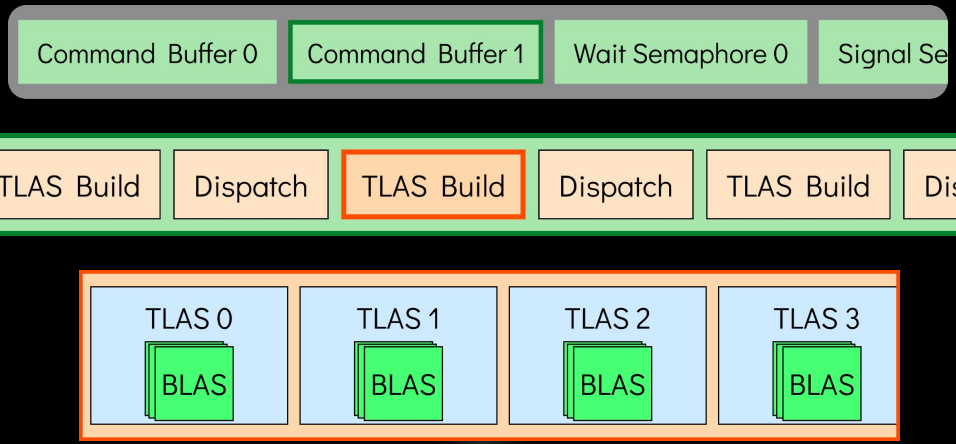


BLAS Lifetime Tracking by TLAS Instances

- Use set<shared_ptr<BLAS>> throughout
  - TLAS can instance a BLAS dozens of times
  - Avoids atomic operation spam by ref. counting

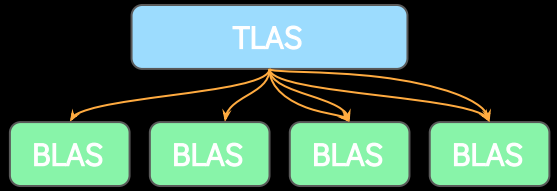


- Record the BLAS sets and target TLAS as metadata of the Build Command !
  - Retain each build's BLAS sets even though only last TLAS build swaps the set
    - *Transiently used BLASes stay alive*



BLAS Lifetime Tracking by TLAS Instances

- Use set<shared_ptr<BLAS>> throughout
  - TLAS can instance a BLAS dozens of times
  - Avoids atomic operation spam by ref. counting



- Record the BLAS sets and target TLAS as metadata of the Build Command !
  - Retain each build's BLAS sets even though only last TLAS build swaps the set
    - *Transiently used BLASes stay alive*

How and When to swap !?

Remember this slide? -CUDA stream host callbacks

- Implemented via **Queue**'s built-in **Timeline Event Handler**
  - CUDA code example:

```
Stream s1;
```

⇔ a single "timeline"

```
// push kernel to queue/stream and continue
gpu_kernel<<<grid, block, s1>>>(data);
```

⇔ vulkan Submit of compute work

```
// Call "MyHostFunction" when gpu
// is done executing the kernel
add_host_callback(s1, MyHostFunction);
```

⇔ **Latching** a callback to **TimelineEventHandler** with the previous submit's signal semaphore!

Remember this slide? -CUDA stream host callbacks

- Implemented via **Queue**'s built-in **Timeline_Event_Handler**

-Atomic TLAS Build Versioning
&
Swap Sets in Submission Callback

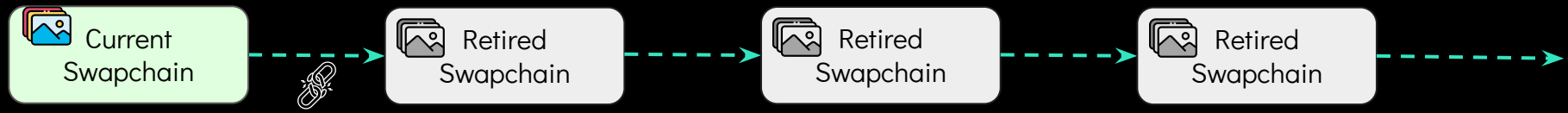
```
add_host_callback(s1, MyHostFunction);
```

Your favourite zombie - Swapchain Destruction !

- You can't destroy a swapchain until all Presentations are complete
  - This includes a *retired* swapchain
  - But there's no function to query that a Presentation is finished
    - Problem that took years to fix, solved with KHR_swapchain_maintenance1
    - Previous agreed upon solution was vkDeviceWaitIdle but still UB

- Retired Swapchain should be done presenting if we acquire enough images on the New

- Connect Acquire and Present to Timeline Semaphores with Empty Submits
  - Binary Semaphores and Fences hidden and managed inside the Swapchain Wrapper



Link breaks after N image acquires

See us at the DevSH table & Get your Merch!

Want to work side-by side with Experts?

- Had your team member poached by a Stealth Mode AI Startup?
- Your organisation hasn't allocated the resources to replace them?
- Talk about us to your Project Manager
    - Or mention us in your trip report!

Visit our website ->

Questions?