Vulkanised 2025

The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

libGPULayers: Diagnostic Vulkan layers for Android

Pete Harris, Arm

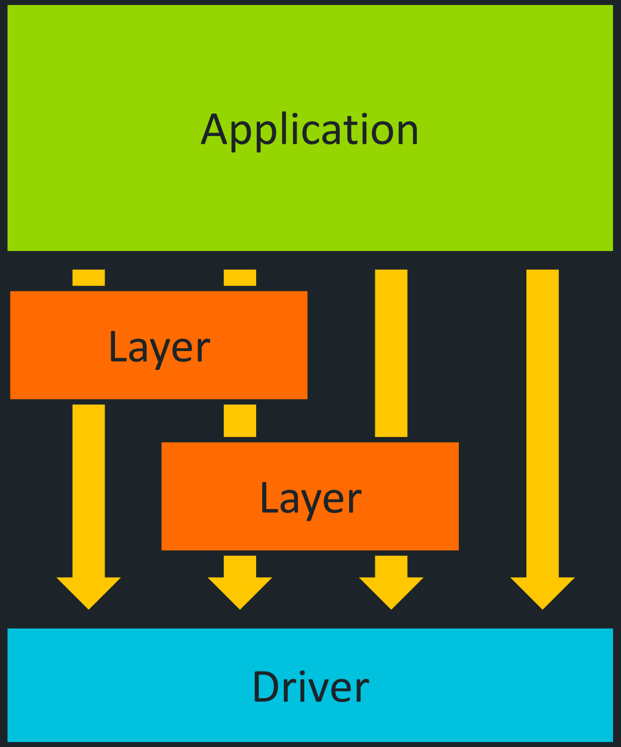
Topics …

- What are layer drivers?
- What is libGPULayers?
- What can it do?
- Layer development thoughts
- Layer API thoughts

Layer driver essentials

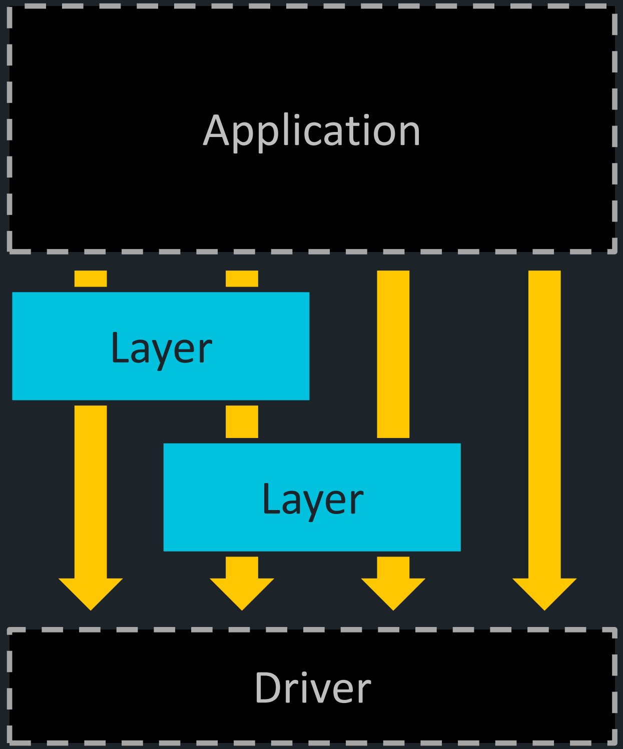
Vulkan layers

- Standard mechanism to inject tooling
  - Orchestrated by the Vulkan loader
- Can monitor application calls
  - E.g., Khronos validation layer
- Can emulate new functionality
  - E.g., Khronos timeline semaphore layer
- Can modify application behavior
  - E.g., most developer tools



We ❤ layer drivers

- Layers are very useful tools!
- We don’t build applications
- We don’t build production drivers
- Layers let us investigate closed systems
  - *Android device must be in developer mode
  - *Android application must be debuggable



Why libGPULayers?

- Experimental layers are great for tech support
  - Investigate what an app does
  - Build a layer to test hypothesis
  - Build a layer to test a fix

- … but making layers by hand is tedious
  - Many lines of boilerplate code
  - Debugging is a pain when they don’t work

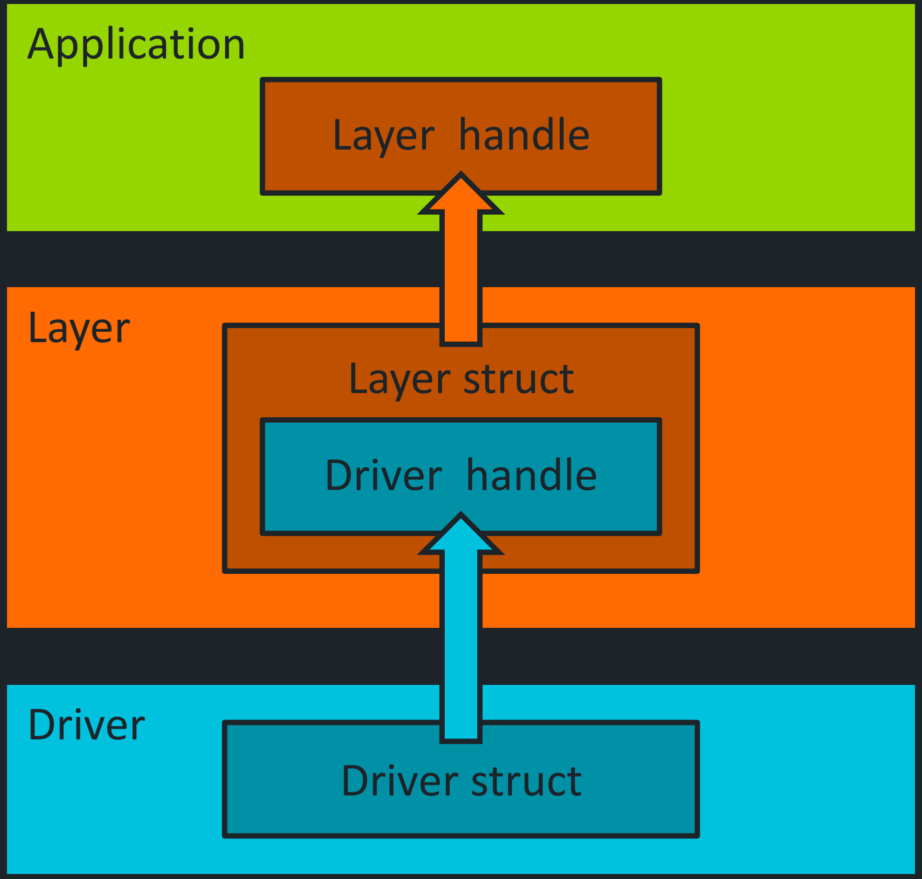
- **Goal #1:** Provide tools to automate layer creation

- **Goal #2:** Provide developers with off-the-shelf layers for remote support

Layer types

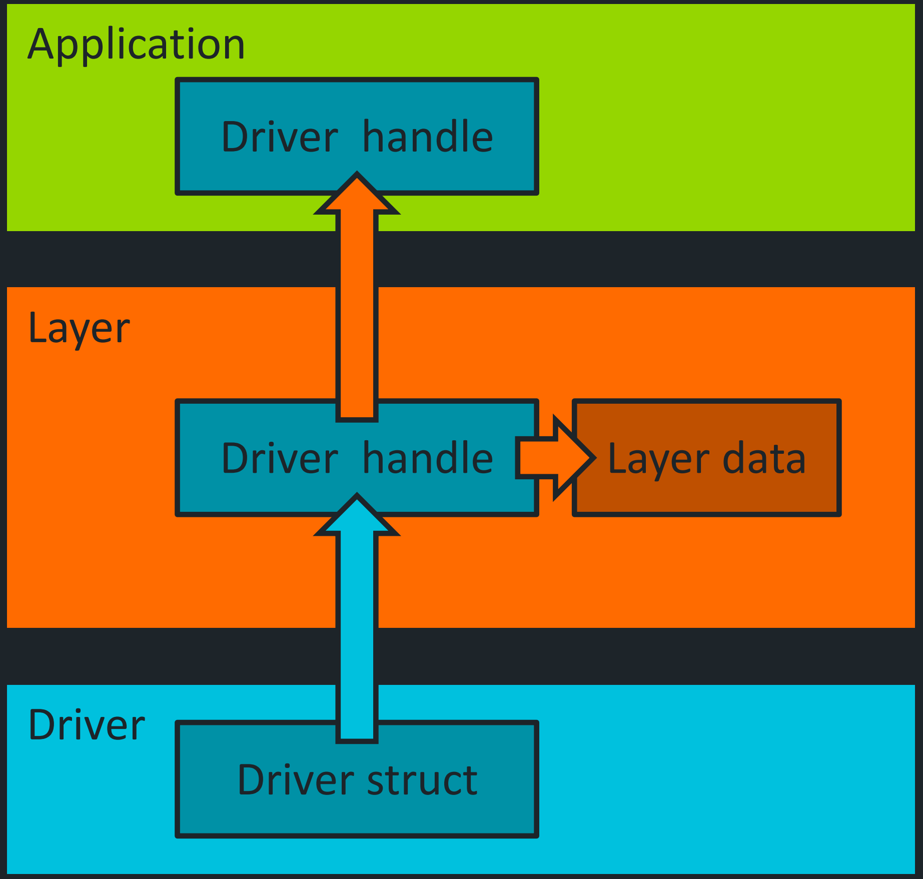
Wrapping layers

- Layer wraps all API objects
  - Application gets layer handles
- **Pros:**
  - Can do 1:N object mapping
  - Dispatch is more efficient
- **Cons:**
  - MUST intercept every use of handles
  - More code to write
  - More fragile



Forwarding layers

- Layer allocates sideband object data
  - Application gets driver handles
  - Layer uses dispatchable handle for lookup
- **Pros:**
  - Can intercept API subset
  - More robust to API updates
- **Cons:**
  - Dispatch is less efficient
  - 1:N object mapping is harder

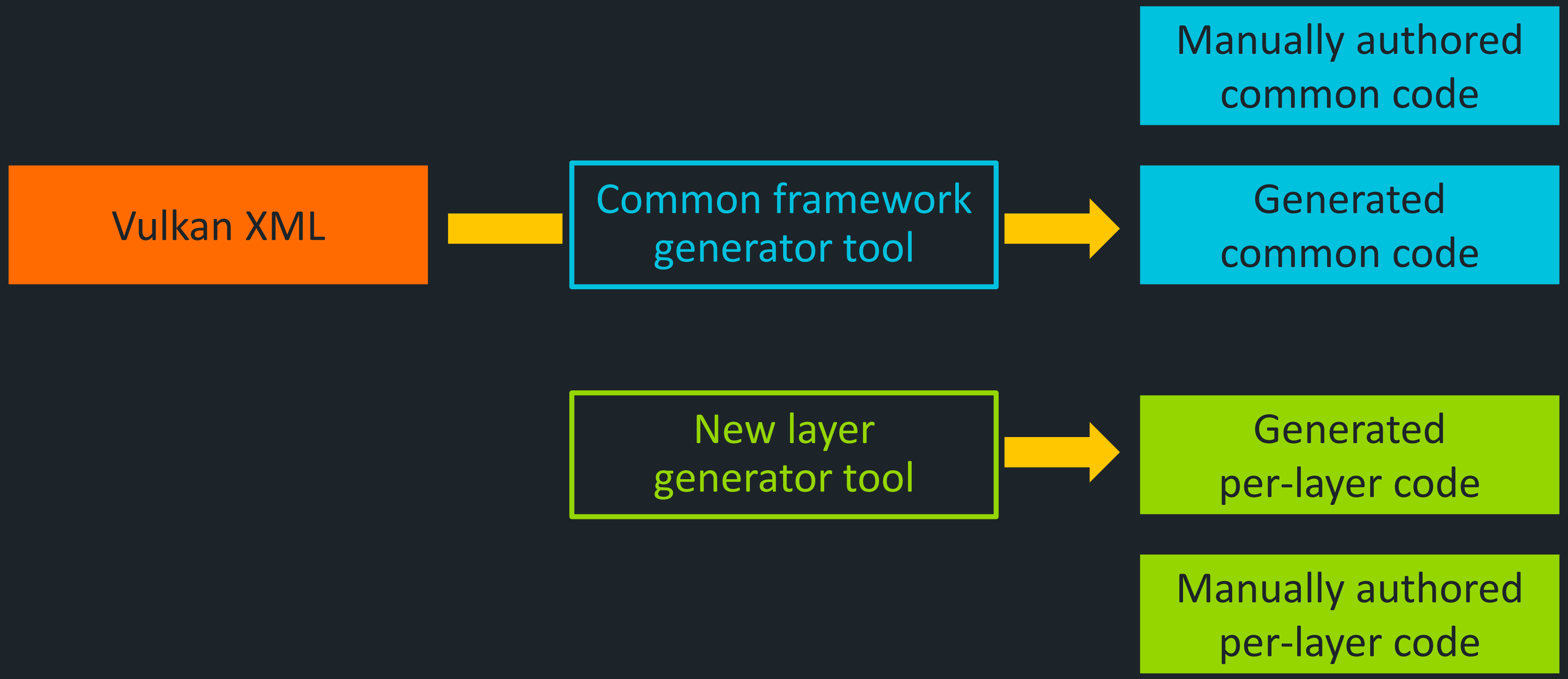


Generating skeleton no-op layer

Generating code

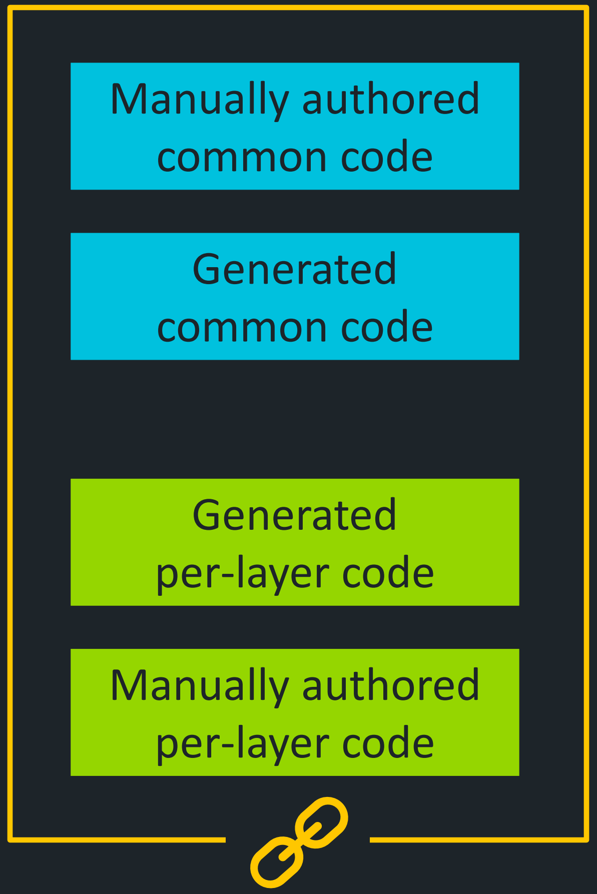
- Layer creation is ideal problem for automation
  - Thousands of lines of boiler plate
  - Machine readable XML specification
- **Goal #1:** Developers write C++ code
- **Goal #2:** Easy to merge Vulkan API updates
- **Goal #3:** Prioritize developer iteration time over run-time

Generating code



Manually authored common code
Vulkan XML
Common framework generator tool
Generated common code
New layer generator tool
Generated per-layer code
Manually authored per-layer code

Building code


Manually authored common code
Generated common code
Generated per-layer code
Manually authored per-layer code

Generating intercept tables

- Function tables are generated from spec XML
  - Need updating when Vulkan API changes
- … but also need modifying per layer
  - Need updating to reflect layer-specific intercepts
- **Risk:** Merge pain!
- **Solution:** C++ templates with tag dispatch
  - Common code provides default implementation
  - Layer code provides specialized implementation
  - Linker does the heavy lifting

Common code

```
struct user_tag {};

struct dispatch_table {
    .vkFoo = vkFoo<user_tag>()
}
```

```
template <typename T>
void vkFoo(…) {
    // Pass-through to driver
}
```

Per-layer code

```
template <>
void vkFoo<user_tag>(…) {
    // Layer implementation
}
```

Standard Android support utilities

Automate the build and platform setup too!

- Android build script provided
  - Just set path to your NDK install
- Android configure script provided
  - Installs layers
  - Configures Android loader
  - Capture logcat (optional)
  - Capture a Perfetto trace (optional)

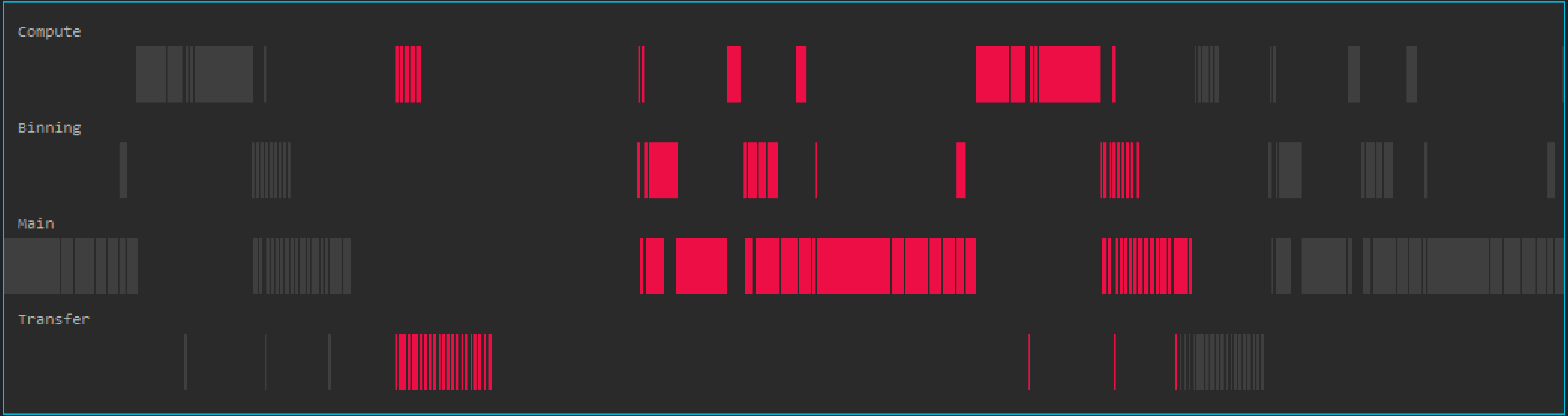
Arm provided layers

#1: GPU support layer

- Layer designed to help with support cases
  - Rendering artefacts
  - DEVICE_LOST errors
- Configurable set of common “does this help?” experiments
  - Force serialize queue and command buffers
  - Force strip shader relaxed precision
  - Force enable/disable framebuffer compression
- Expect to grow over time
  - We will package up things we find useful

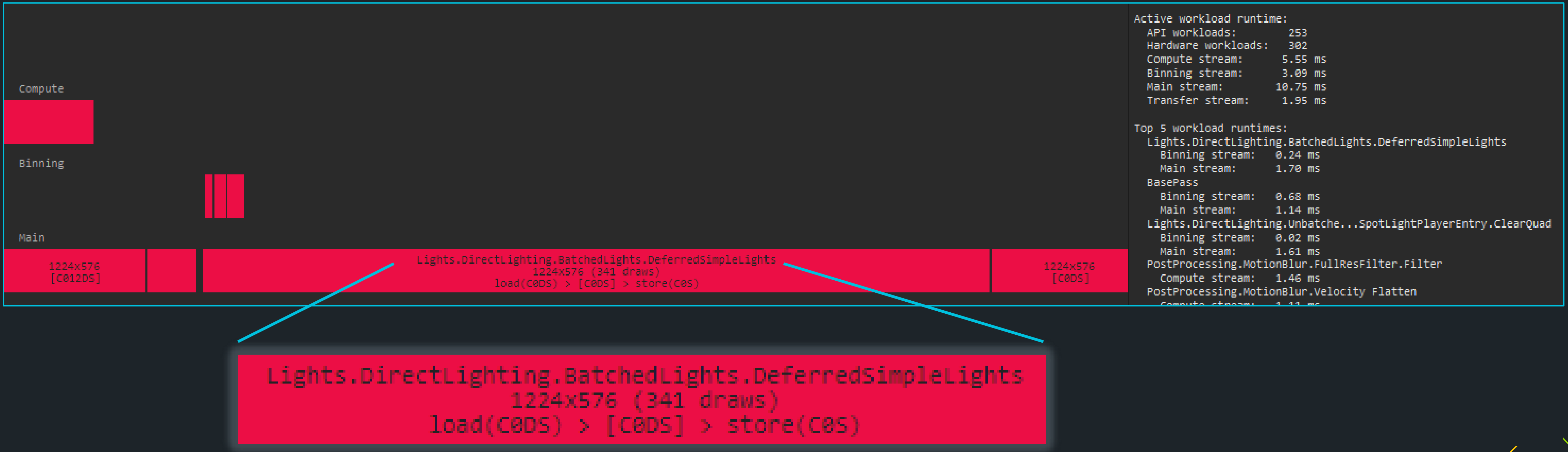
#2: GPU timeline layer

- Layer designed to annotate Perfetto Render Stages traces
  - For example, our Unreal Engine-based tech demo
  - 253 API workloads per frame, 302 hardware workloads per frame
  - What are they? What are they doing?



#2: GPU timeline layer

- Layer exports semantic metadata via side-channel
  - Tags workloads with a unique debug label
  - Emits metadata packet associated with each tag
  - Experiential viewer is included!



Compute
Binning
Main
1224x576
[C012DS]
Lights.DirectLighting.BatchedLights.DeferredSimpleLights
1224x576 (341 draws)
load(C0DS) > [C0DS] > store(C0S)
1224x576
[C0DS]

```
Active workload runtime:
  API workloads:         253
  Hardware workloads:    302
  Compute stream:      5.55 ms
  Binning stream:      3.09 ms
  Main stream:        10.75 ms
  Transfer stream:     1.95 ms

Top 5 workload runtimes:
  Lights.DirectLighting.BatchedLights.DeferredSimpleLights
    Binning stream:   0.24 ms
    Main stream:      1.70 ms
  BasePass
    Binning stream:   0.68 ms
    Main stream:      1.14 ms
  Lights.DirectLighting.Unbatche...SpotLightPlayerEntry.ClearQuad
    Binning stream:   0.02 ms
    Main stream:      1.61 ms
  PostProcessing.MotionBlur.FullResFilter.Filter
    Compute stream:   1.46 ms
  PostProcessing.MotionBlur.Velocity Flatten
```

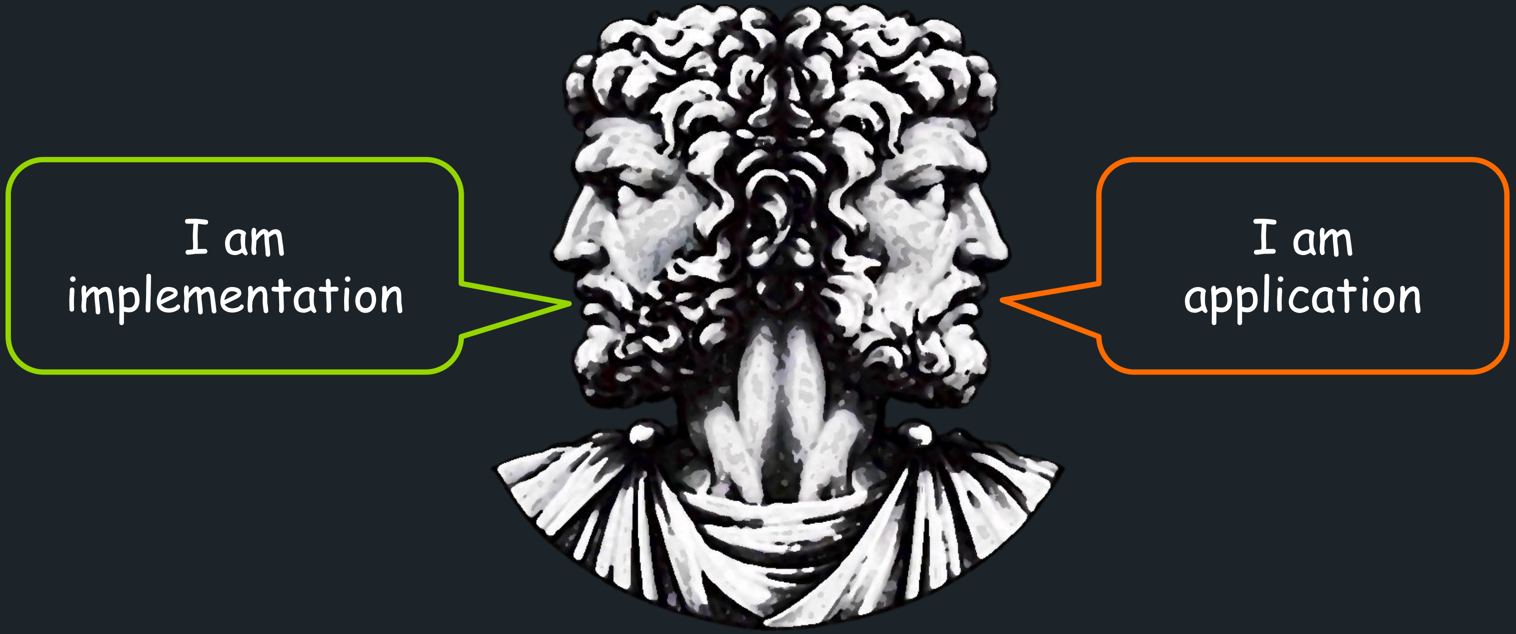
#3: GPU performance layer
(WORK IN PROGRESS)

- Layer designed to help with performance measurement
  - Per frame performance counters
  - Per workload performance counters
  - Per workload timer queries
- Serializes around measurement points
  - Aim to measure the workload cost
  - Need to stop tile-based rendering overlapping things!
- **Future:** On-screen per-frame metrics overlay
  - Live overlay for common performance measures

Development thoughts

Layers are an odd fit for Vulkan

- **Vulkan:** “I am an application API”

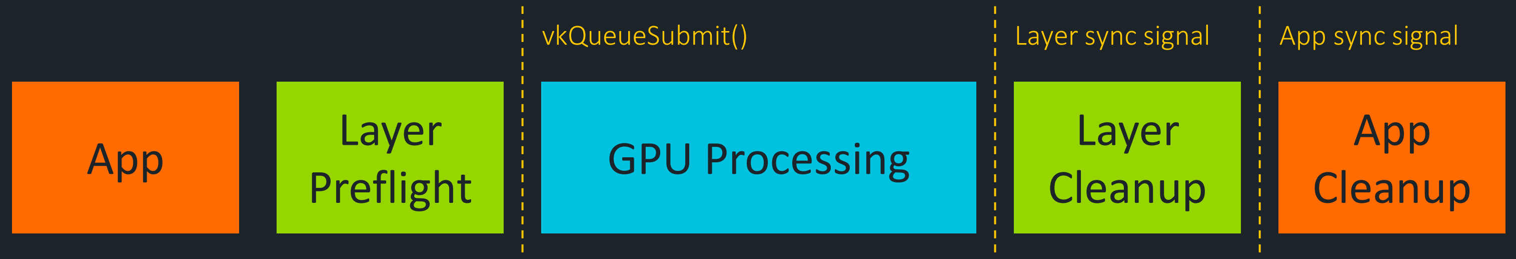


- **Layer:** “… but …”

Resource lifetime gotchas

Common use case is to instrument submits

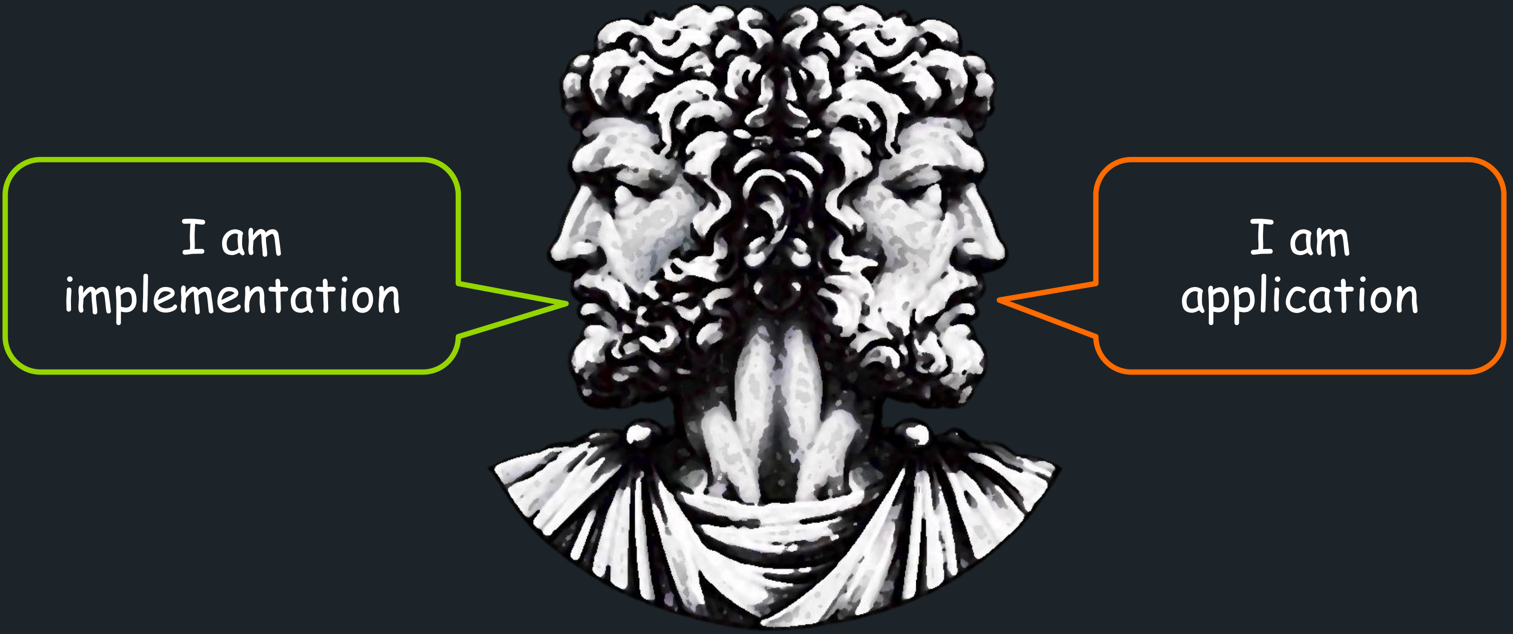
- Ideal pattern for layer developers is an onion



- … but it’s exceptionally hard to build in practice
  - Must virtualize every GPU-to-CPU synch
  - Must provide software implementations of most of them

Layers are an odd fit for Vulkan

- **Vulkan:** “You know your resource lifecycle …”

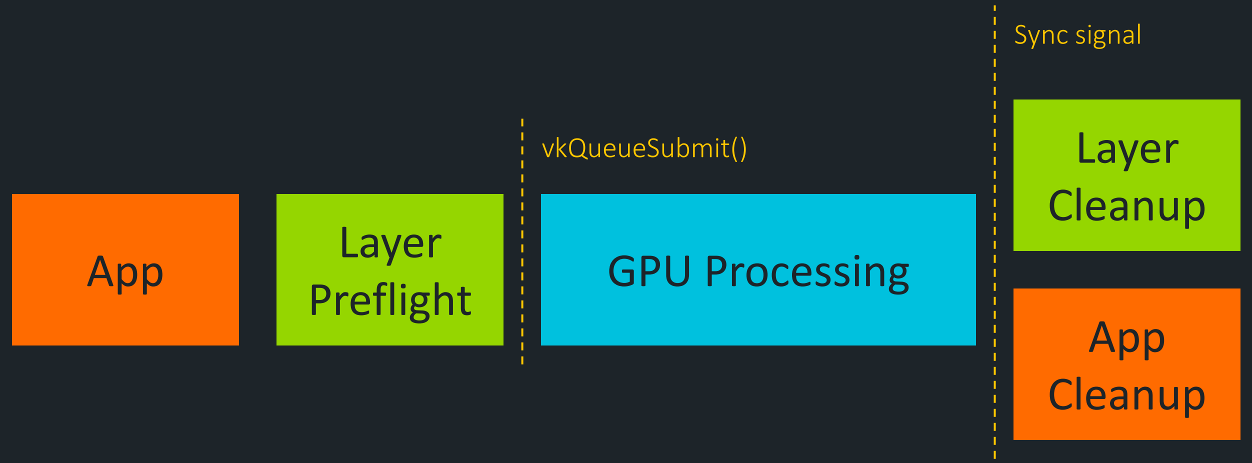


- **Layer:** “… but …”

Resource lifetime gotchas

Common use case is to instrument submits

- Easy implementation is therefore a forked cleanup



- … but cleanup is now racy!
  - Don’t tie layer resources to the application resource lifetime!
  - Ref-count layer resources like you are an OpenGL ES driver …

Layers are an odd fit for Vulkan

- **Vulkan:** "You know your workloads …"

I am implementation
I am application

- **Layer:** "… but …"

Workload instrumentation

- Not all workload state is pre-recorded in the command buffer
  - Dynamic render passes resolved at submit time
  - Debug marker label stack resolved at submit time
  - Indirect parameters resolved at runtime

- … but what the layer needs to do *is* defined by the command buffer

- **Design pattern:** Software command buffers
  - Recorded alongside API command buffer
  - Preflight command stream executed before vkQueueSubmit()
  - Resolve command stream executed asynchronously based on API sync triggers

Development API niggles

Android loader is basic

- Ideally a layer can query what is available beneath it!
  - Is the API version new enough for the layer?
  - Are the necessary extensions available?

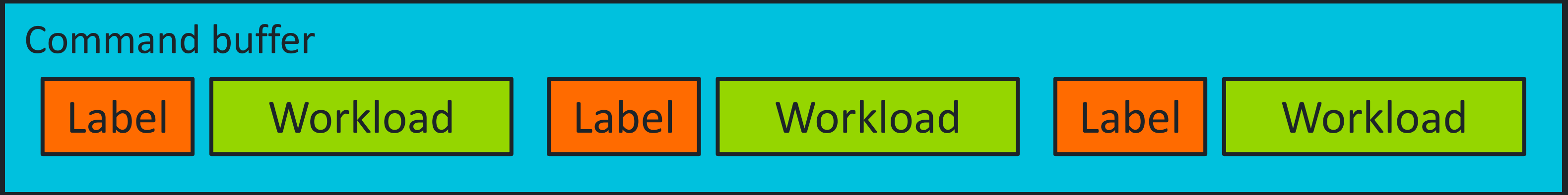
- Android loader implements the v0 loader interface
  - Does not support chain calling pre-instance functions
  - No vkEnumerateInstanceVersion() for API version
  - No vkEnumerateInstanceExtensionProperties() for instance extensions

- It will work if you are bottom layer in the stack
  - … but not if there are other layers beneath you

Command buffer instrumentation pain points

Workload identification

- We want to instrument specific workloads
  - Must identify individual workloads inside a command buffer
- Inject debug markers in the command buffer

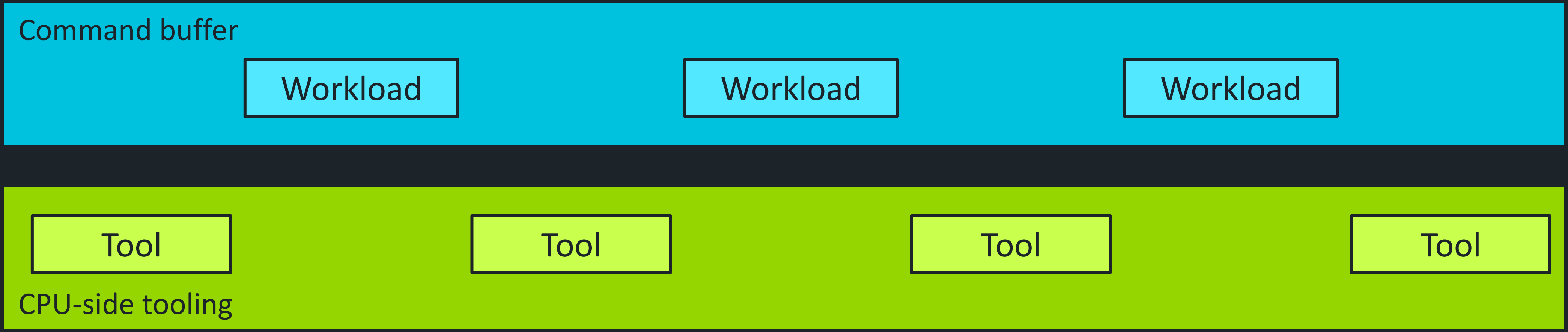


- Command buffer instrumentation is fixed at record time
  - **Problem:** Multi-submit command buffers make tools sad

Command buffer instrumentation pain points

CPU traps

- We want to instrument specific workloads
  - Not all our tooling is accessible from the command stream

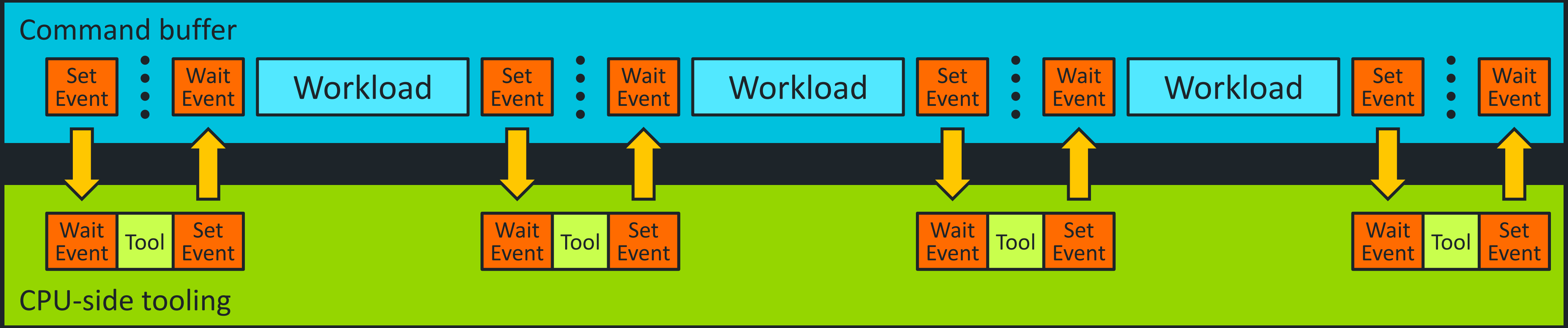


- Proper solution:
  - Split command buffers into one per workload
  - Complex, with a high software cost

Command buffer instrumentation pain points

CPU traps

- Our current nasty hack …
  - (Yes, it's out of spec)



Find out more on GitHub

- Make your own layers quickly!
- Use our off-the-shelf layers to diagnose common problems quickly!
- Use our off-the-shelf layers to customize data visualization in other Arm tools
  - (Future looking statement …)

github.com/ARM-Software/libGPULayers

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה