Vulkan Introduction

05 / 29 / 18

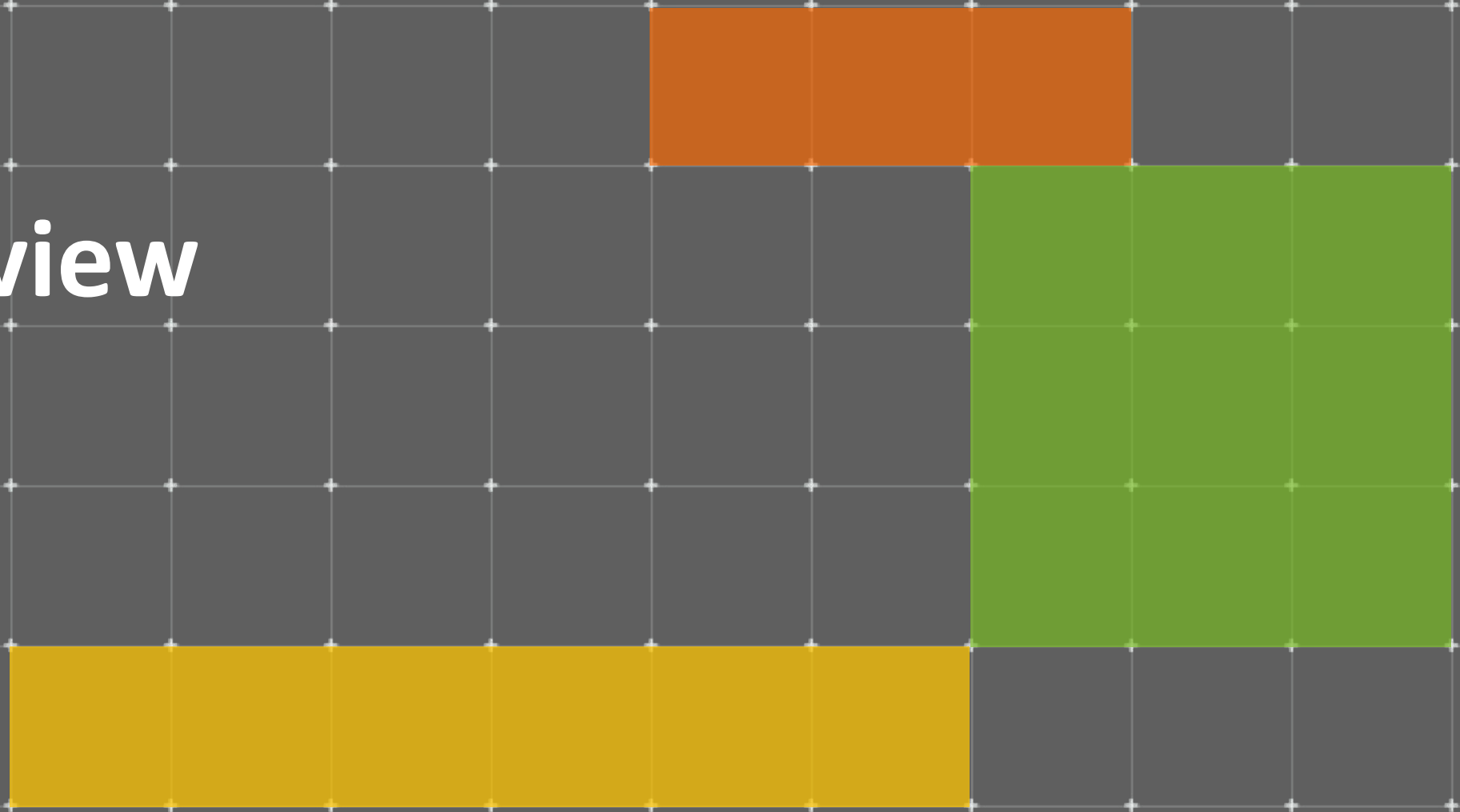
PEG, Partner Enablement Group
Mali Media support

Goal for this section

Explain what Vulkan brings to the table for mobile graphics over OpenGLES, and how it can improve your renderers.

Not going into deep detail, only focusing on high-level API features.

Overview



arm

Why Vulkan exists

The industry has expressed a desire for explicit control over the GPU.

- Experience with the level of control you have in console APIs.

This push materialized into a new suite of modern APIs the last years:

- Direct3D 12 (Windows 10 only)
- Vulkan (Cross-platform)
- Metal 1, Metal 2 (Apple only)

Vulkan evolved from the AMD's Mantle API.

Some problems with older APIs

OpenGL and OpenGLES have several long-standing issues which are considered unfixable.

OpenGL is single threaded

- As 4 and even 8 cores are present in even most smartphones these days, not having freely threaded graphics APIs is not good enough

The API model is very far away from how hardware works

- The OpenGL abstraction models are very foreign to hardware, and this translation comes at a significant CPU cost

The API does not express the asynchronous nature of GPUs

- GPUs are highly asynchronous, and the older driver models need to spend a lot of effort in making you believe the GPU is synchronous. This causes highly complex and unpredictable driver behavior.

Very hard to reason about how to hit good paths in the driver

- This tends to be extremely vendor and driver specific.

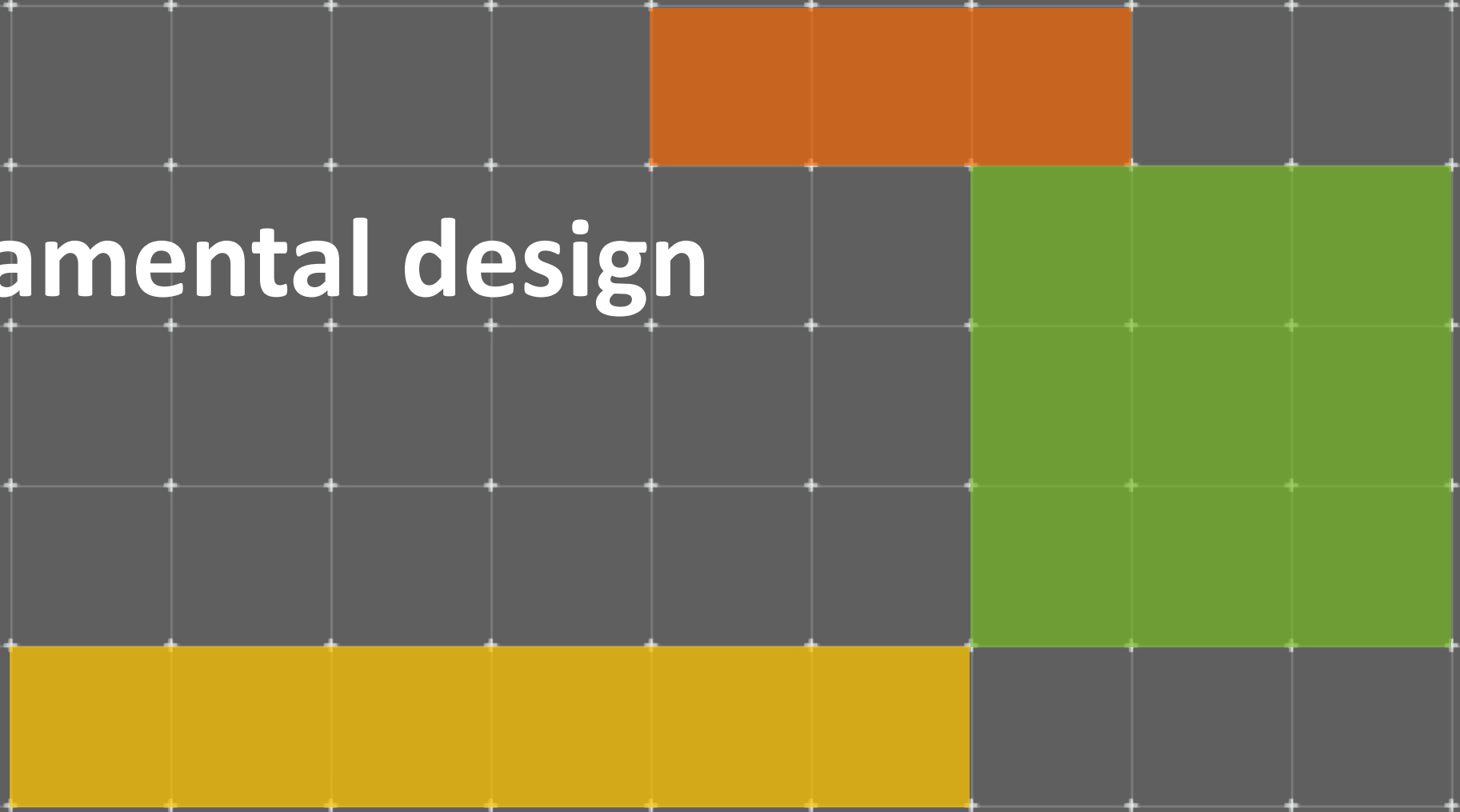
Some OpenGLES problems for Mali

OpenGLES is derived from desktop OpenGL, and thus has a model closely resembling desktop hardware.

There is no concept of render passes. Drivers will need to determine this heuristically.

No standard way of exposing tile-memory.

Fundamental design



arm

Explicit

Vulkan is very explicit and verbose. In older APIs, drivers generally have had to deduce information at the very last minute.

- The most common problem here is shader compilation stuttering when using a new render state.

In Vulkan, all information is provided up-front to the driver.

- VkRenderPass
- VkPipeline
- VkPipelineLayout

Removing guess-work from the driver is a huge design win for Vulkan.

Layers and error handling

Error handling is a common frustration among developers. All vendors implement this differently in GLES.

Error handling cannot be turned off, meaning shipping titles with zero errors will still run exhaustive error checking for all cases even in release builds.

In Vulkan, drivers perform little to no error checks. Make a mistake, and you'll likely crash.

Validation layers are provided which you can easily plug into during development and catch a huge range of errors in a vendor-neutral way.

Layers also enable a much easier way to hook application than what you had to do in older APIs (DLL injection, LD_PRELOAD hell, etc ...)

One API for mobile and desktop

While OpenGL and OpenGLES are fairly close, it is still very painful to make a renderer which targets both. Even within the APIs there are several «sub-APIs» which often need to be handled somewhat differently.

- OpenGLES 2.0
- OpenGLES 3.x
- OpenGL 3.x
- OpenGL 4.x with and without AZDO extensions

Vulkan provided one API for both mobile and desktop, making cross-platform development far easier.

Desktop GPUs do generally support a much larger feature-set than mobile, but this is exposed through caps bits in VkPhysicalDeviceFeatures rather than defining a new API.

Multithreading

Vulkan is designed for multithreading. Vulkan commands either fall into externally or internally synchronized.

Internally synchronized commands are typically heavyweight commands which you should never do in performance critical code.

- Allocating VkDeviceMemory – Suballocate from your own heap ☺
- Creating pipelines, etc – This can be done up-front

Externally synchronized is where you as a developer need to make sure there is no race condition.

SPIR-V

Traditionally, shading languages in OpenGL has been string-based, i.e., every vendor needed to implement their own GLSL frontend.

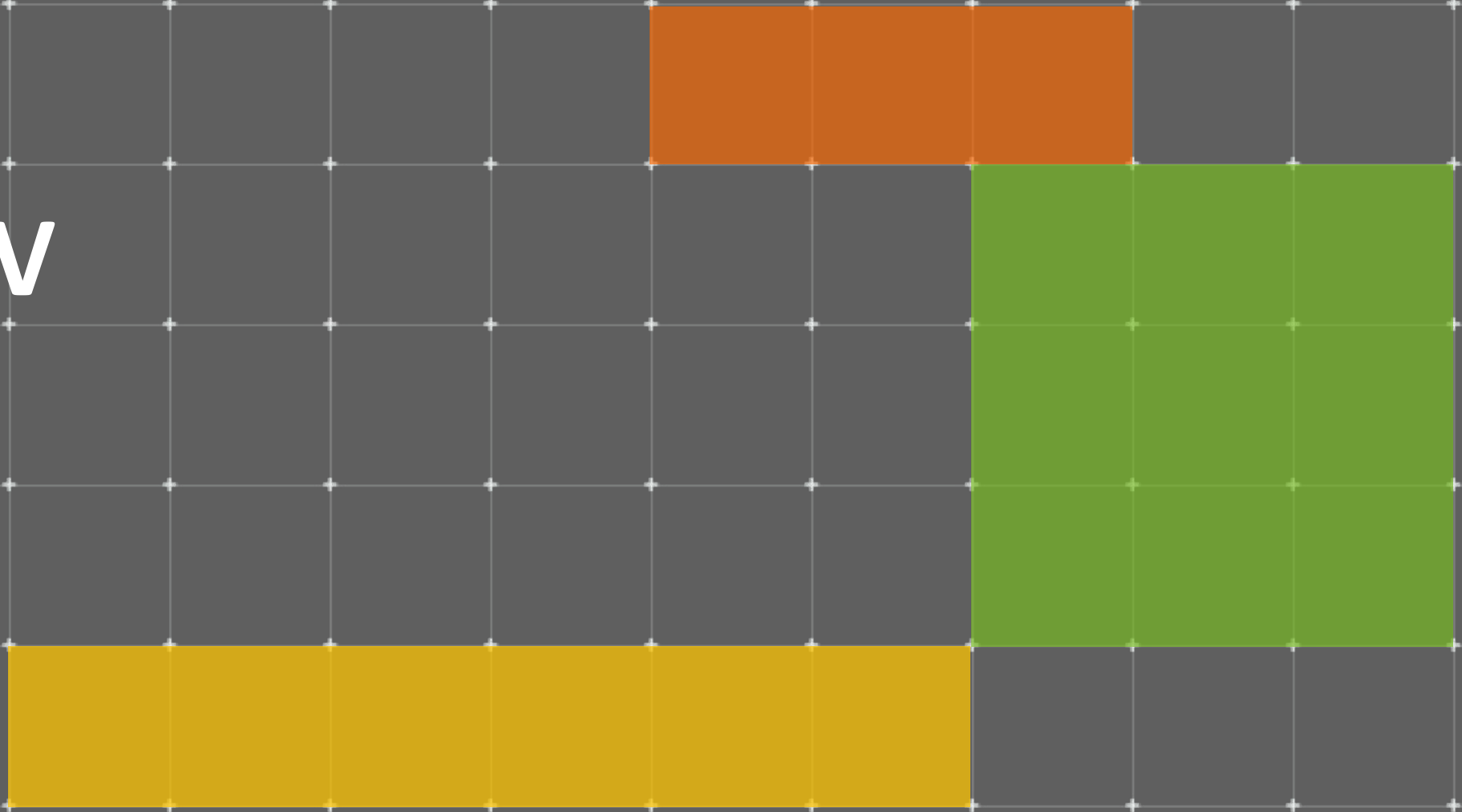
- Many, many silly bugs ☺

SPIR-V is the only shading language format used in Vulkan.

Binary intermediate representation aligns the responsibilities closer to how D3D works.

Freedom in which high-level language to use if you have a SPIR-V target.

SPIR-V



arm

SPIR-V ecosystem

Frontends

- glslangValidator (GLSL and HLSL)
- DirectXShaderCompiler (SPIR-V output being worked on by Google)

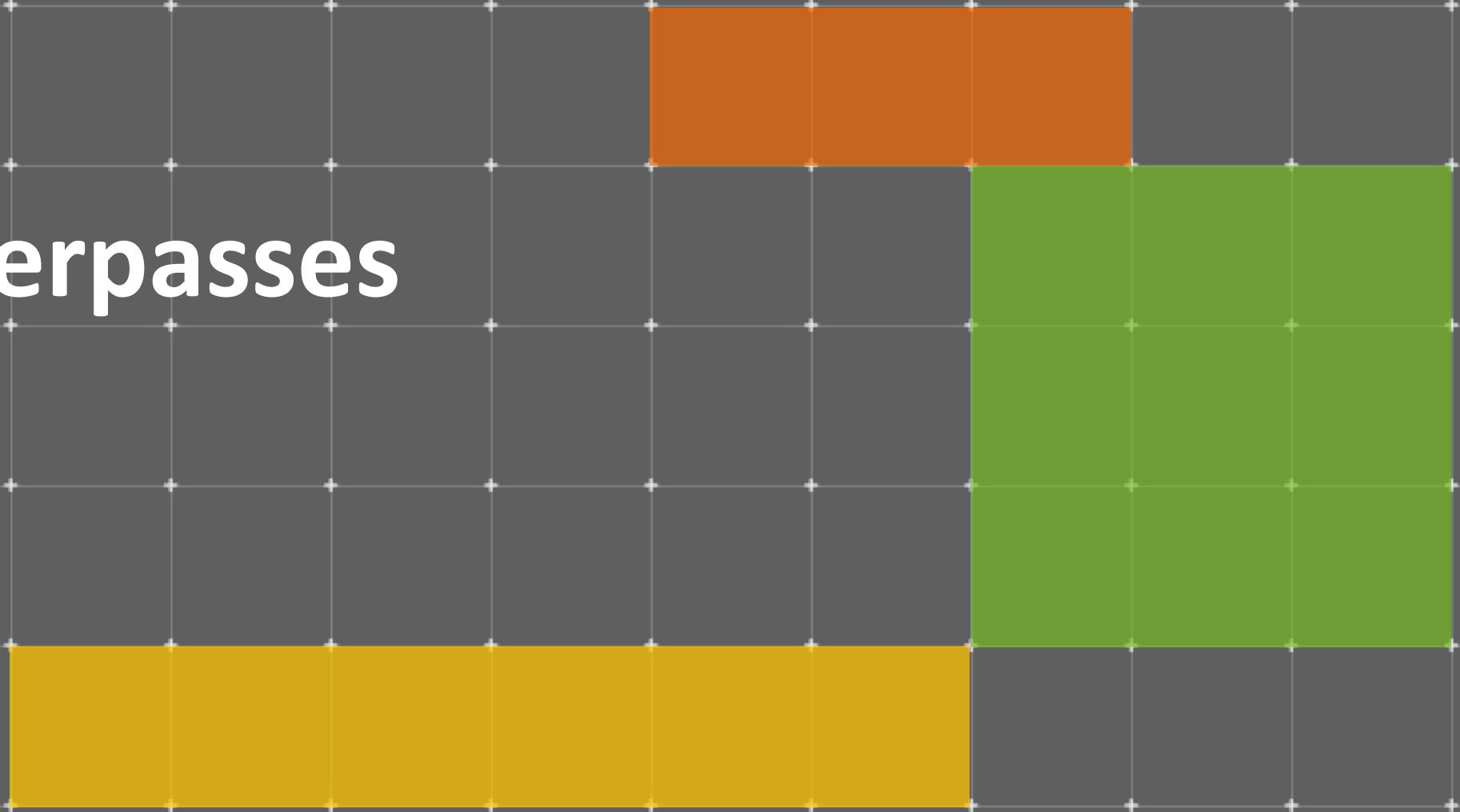
SPIRV-Tools

- Assembler/disassembler
- Validator
- Optimizer

SPIRV-Cross

- Extensive reflection API
- Cross-compilation to GLSL, HLSL, MSL

Renderpasses



arm

Explicit render passes in Vulkan

In Vulkan, the lifetime of a render pass is clearly laid out.

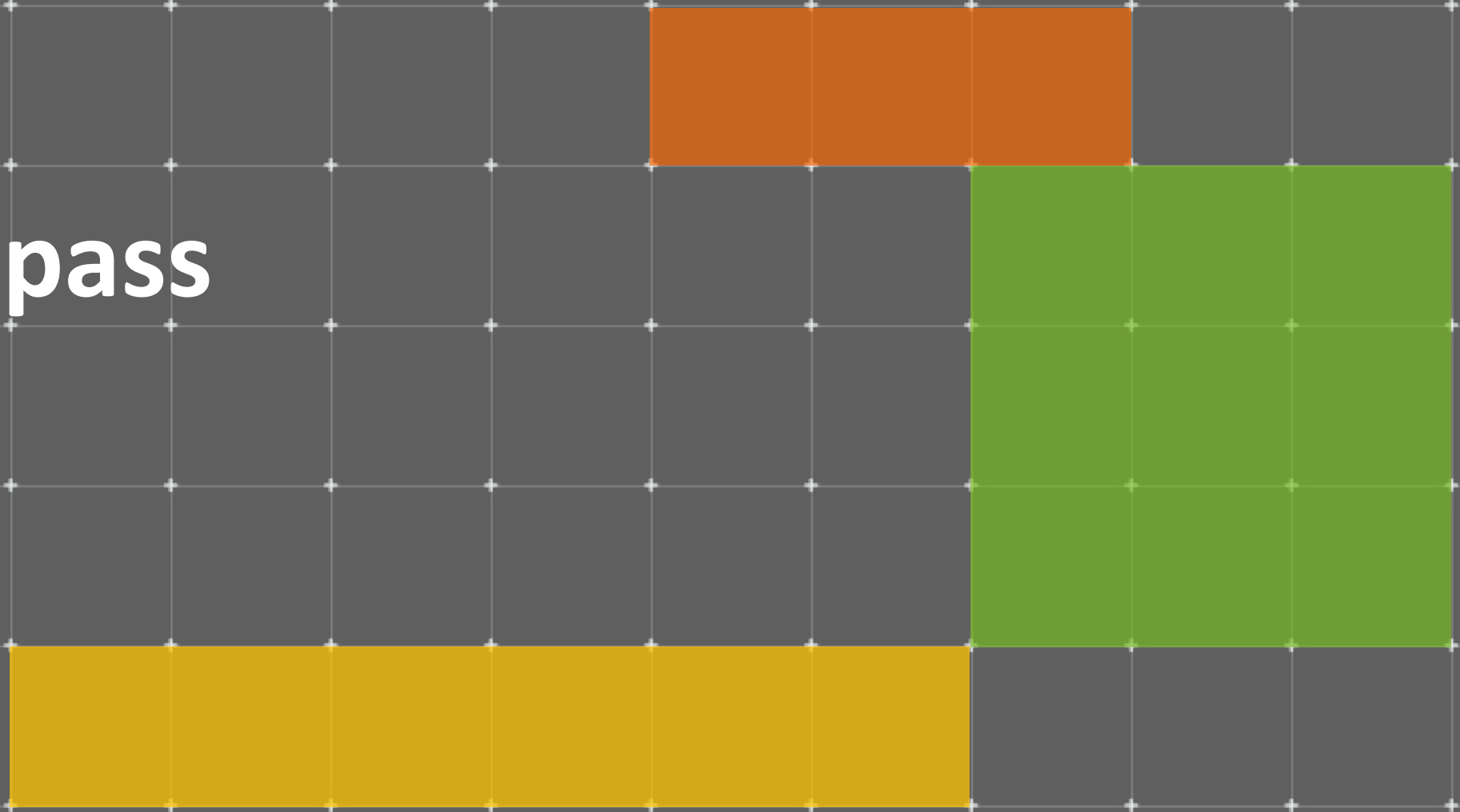
vkCmdBeginRenderPass()

- Starts a render pass, driver knows ahead of time which attachments to clear, which attachments to load from memory into tile-buffer.

vkCmdEndRenderPass()

- Completes a render pass, driver already knows which attachments to flush out to main memory.

Multipass



arm

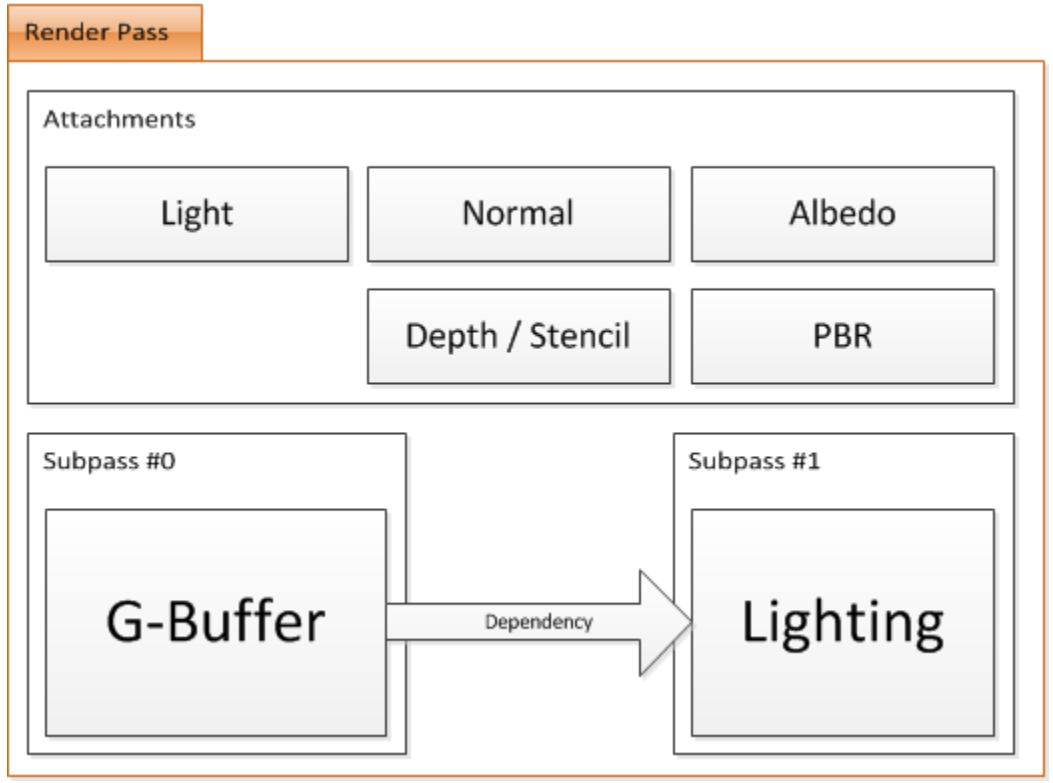
What is Multipass?

Renderpasses can have multiple subpasses

Subpasses can have dependencies between each other

- Render pass graphs

Subpasses refer to subset of attachments



arm

Improved MRT deferred shading in Vulkan

Classic deferred has two render passes

- G-Buffer pass, render to ~4 textures
- Lighting pass, read from G-Buffer, accumulate light

Lighting pass only reads G-Buffer at gl_FragCoord

Rethinking this in terms of Vulkan multipass

- Two subpasses
- Dependencies
  - COLOR | DEPTH -> INPUT_ATTACHMENT | COLOR | DEPTH_READ
  - **VK_DEPENDENCY_BY_REGION_BIT**

Vulkan GLSL subpassLoad()

Reading from input attachments in Vulkan is special

- Special image type in SPIR-V

On vkCreateGraphicsPipelines we know

- renderPass
- subpassIndex

subpassLoad() either becomes

- texelFetch()-like if subpasses were **not** fused
  - This is why we need VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
- magicReadFromTilebuffer() if subpasses were fused

Compiler knows ahead of time

- No last-minute shader patching required

Transient attachments

After the lighting pass, G-Buffer data is not needed anymore

- G-Buffer data only needs to live on the on-chip SRAM
- Clear on render pass begin, no need to read from main memory
- storeOp is DONT_CARE, so never actually written out to main memory

Vulkan exposes lazily allocated memory

- imageUsage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
- memoryProperty = VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
- On tilers, no need to back these images with physical memory ☺

Multipass benefits everyone

Deferred paths essentially same for mobile and desktop

- Same Vulkan code (*)
- Same shader code (*)

VkRenderPass contains all information it needs

- Desktop can enjoy more informed scheduling decisions
- Latest desktop GPU iterations seem to be moving towards tile-based
- At worst, it’s just classic MRT

(*) Minor tweaking to G-Buffer layout may apply


ARM

Baseline test data

Measured on Galaxy S7 (Exynos)

4096x2048 resolution

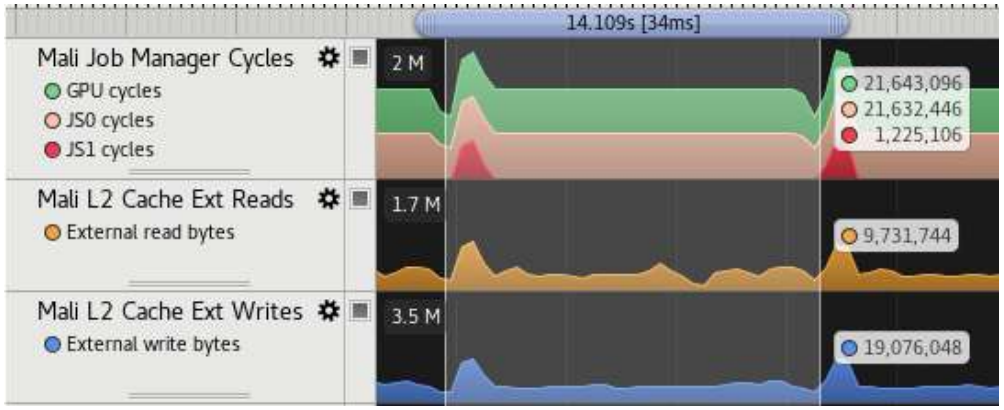
- Hit V-Sync at native 1440p

~30% FPS improvement

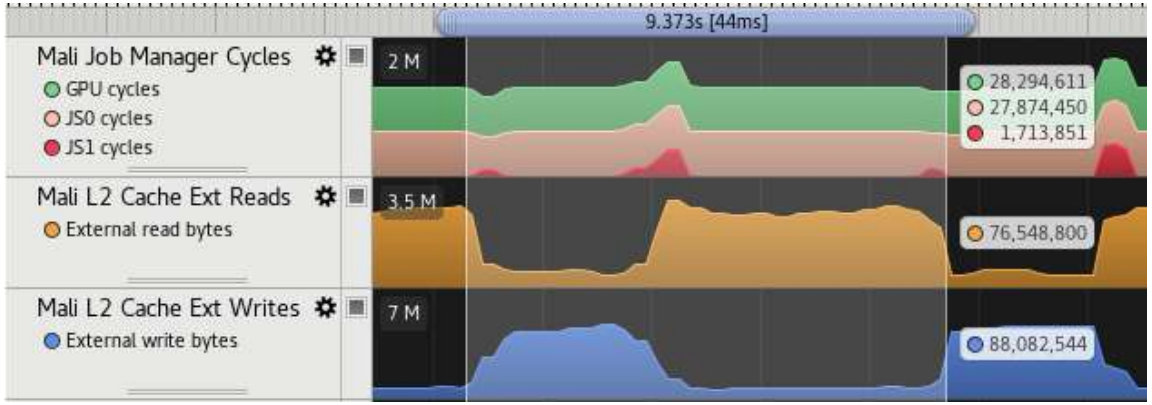
~80% bandwidth reduction

- Only using albedo and normals
- Saving bandwidth is vital for mobile GPUs

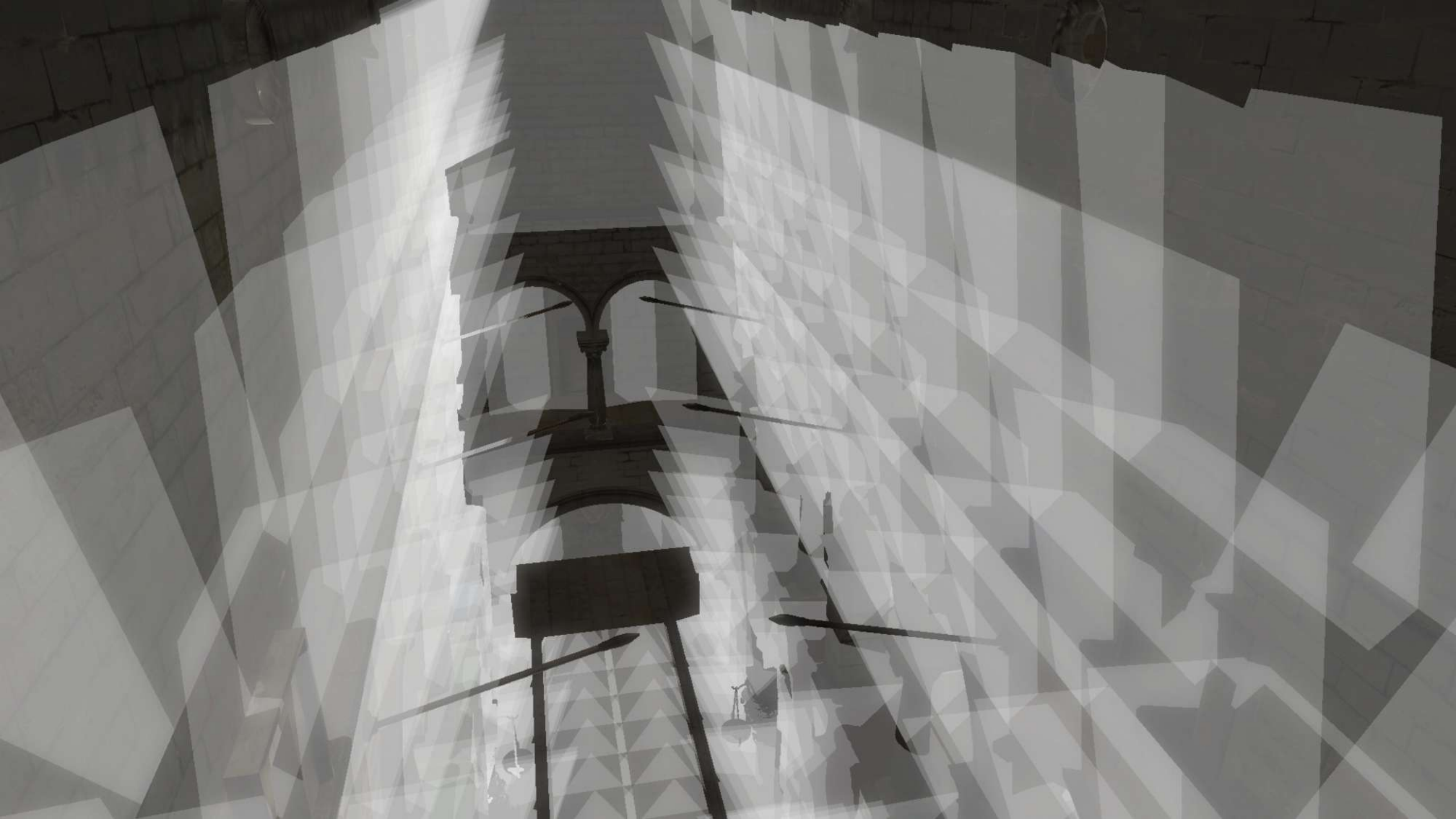
Multipass



MRT



arm



Sponza test data

Far, far heavier than sensible mobile content

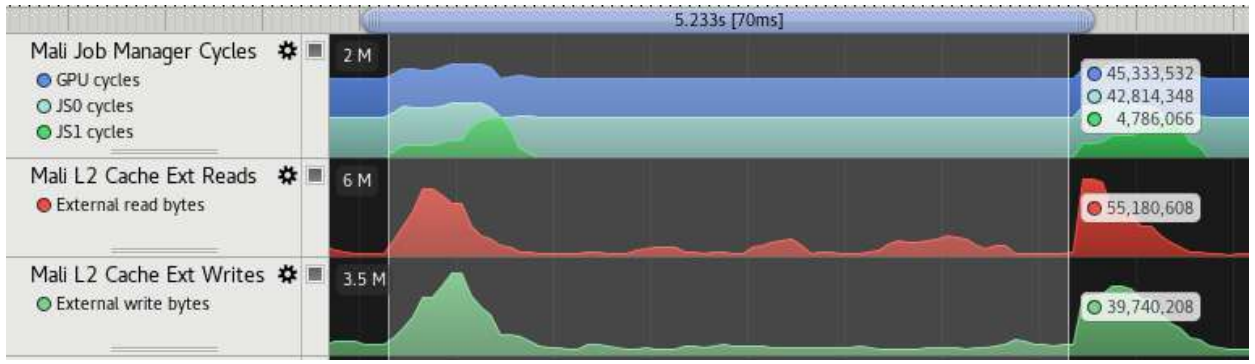
1440p (native)

- Overkill

~50-60% bandwidth reduction

~18% FPS increase

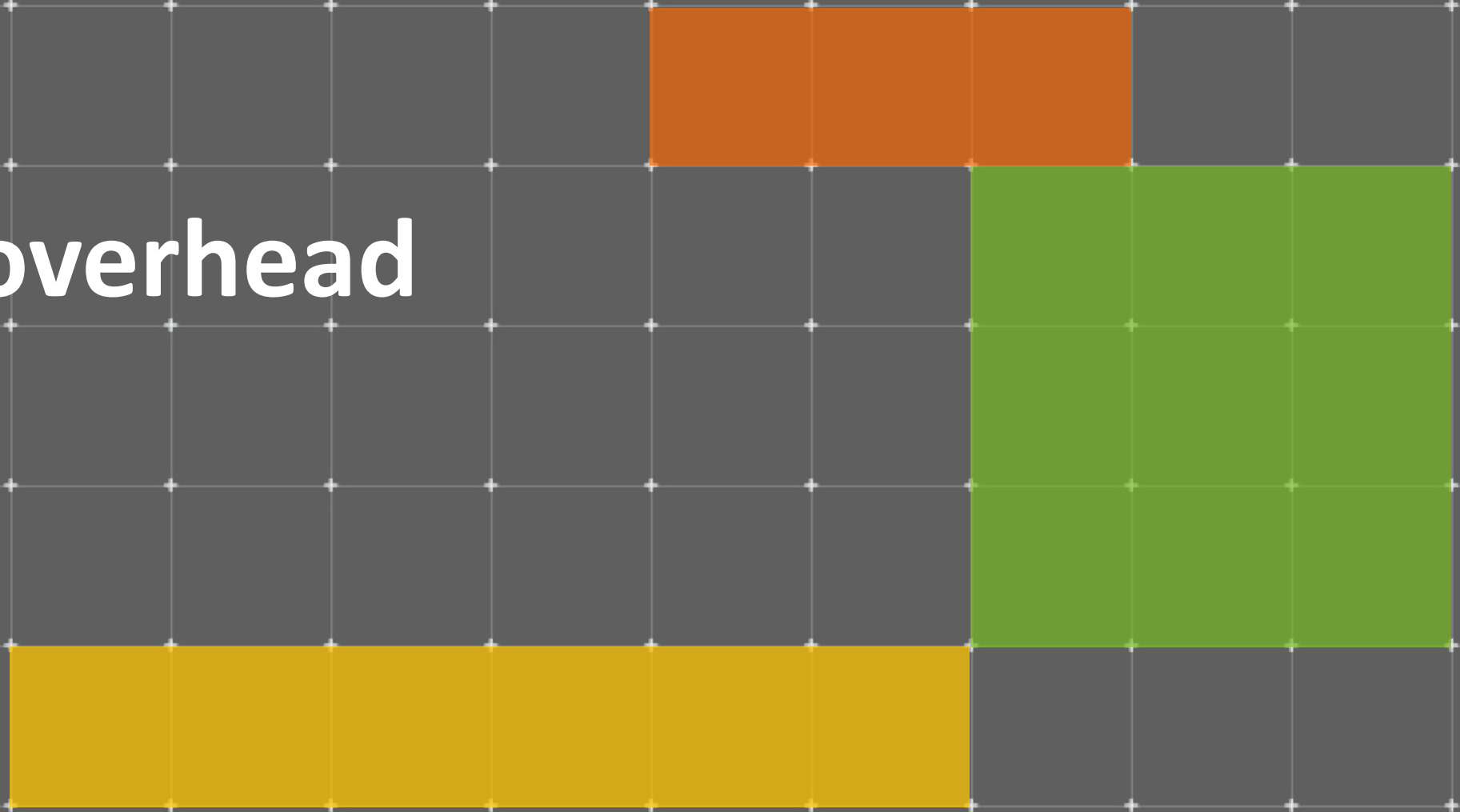
Multipass



MRT

arm

CPU overhead



arm

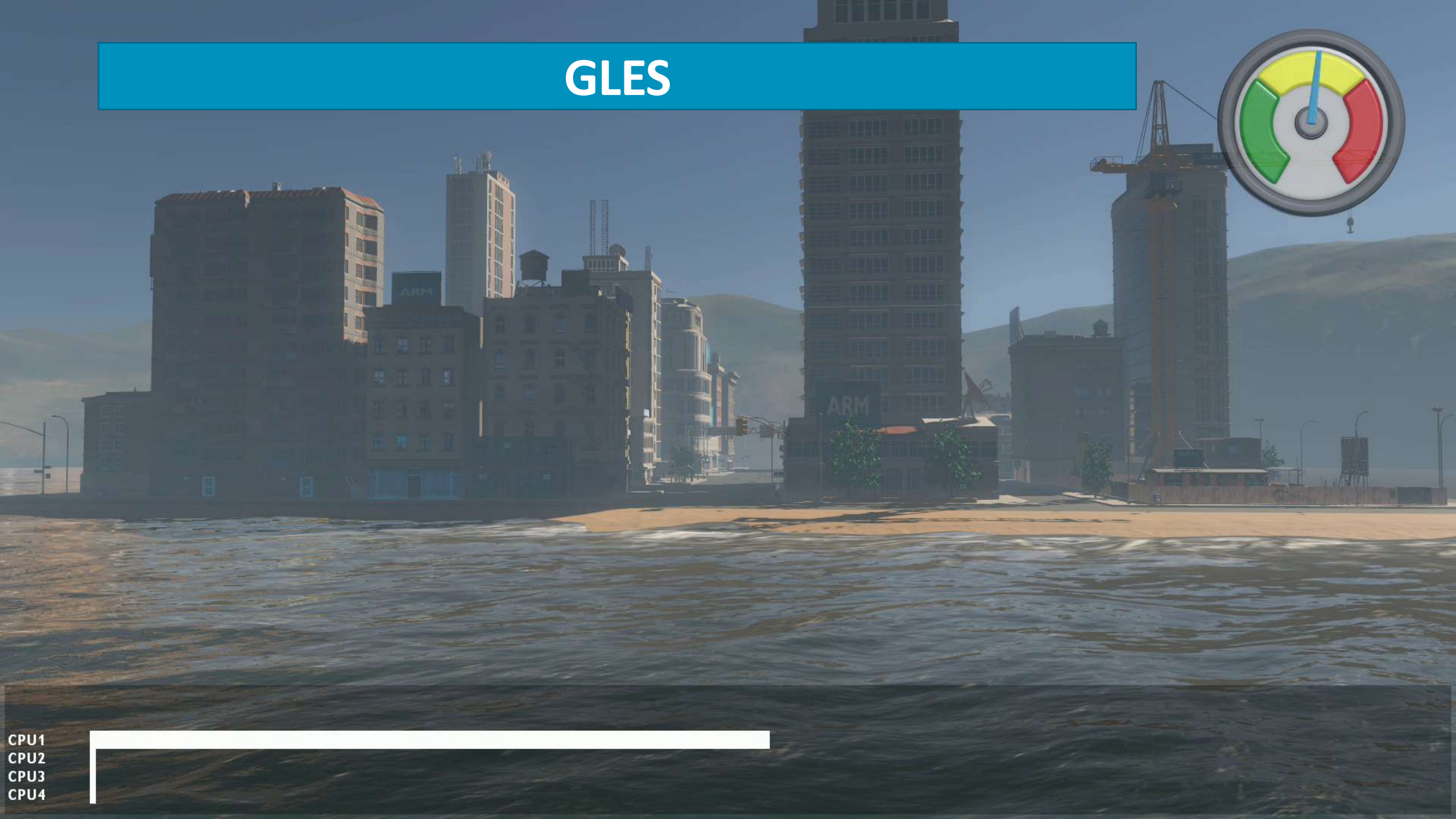
CPU overhead reduction

While OpenGL is pretty alright at driving the GPU, the CPU cost for doing so is quite significant. Many applications struggle with CPU overhead on mobile, and power wasted on API overhead can interfere with GPU clocks.

A good rule of thumb which we have observed in multiple real-world engines is a **3x reduction** in CPU time for single-threaded Vulkan over GLES with fairly straight ports.

With multithreading, we have seen **10x wall-time improvements** in our own content, as well as **15% system-wide power reduction**.

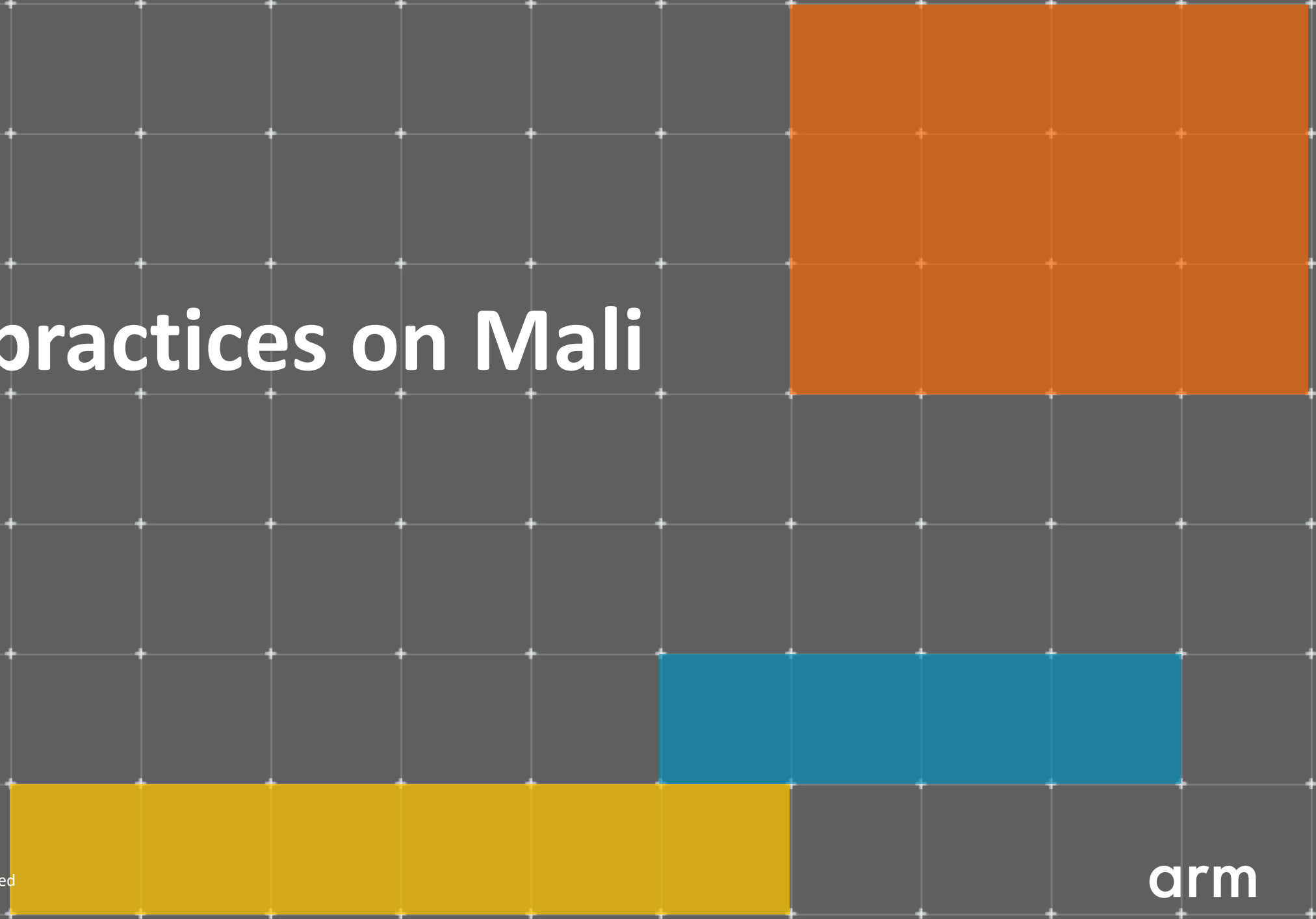
GLES



Vulkan

CPU1
CPU2
CPU3
CPU4

Best practices on Mali



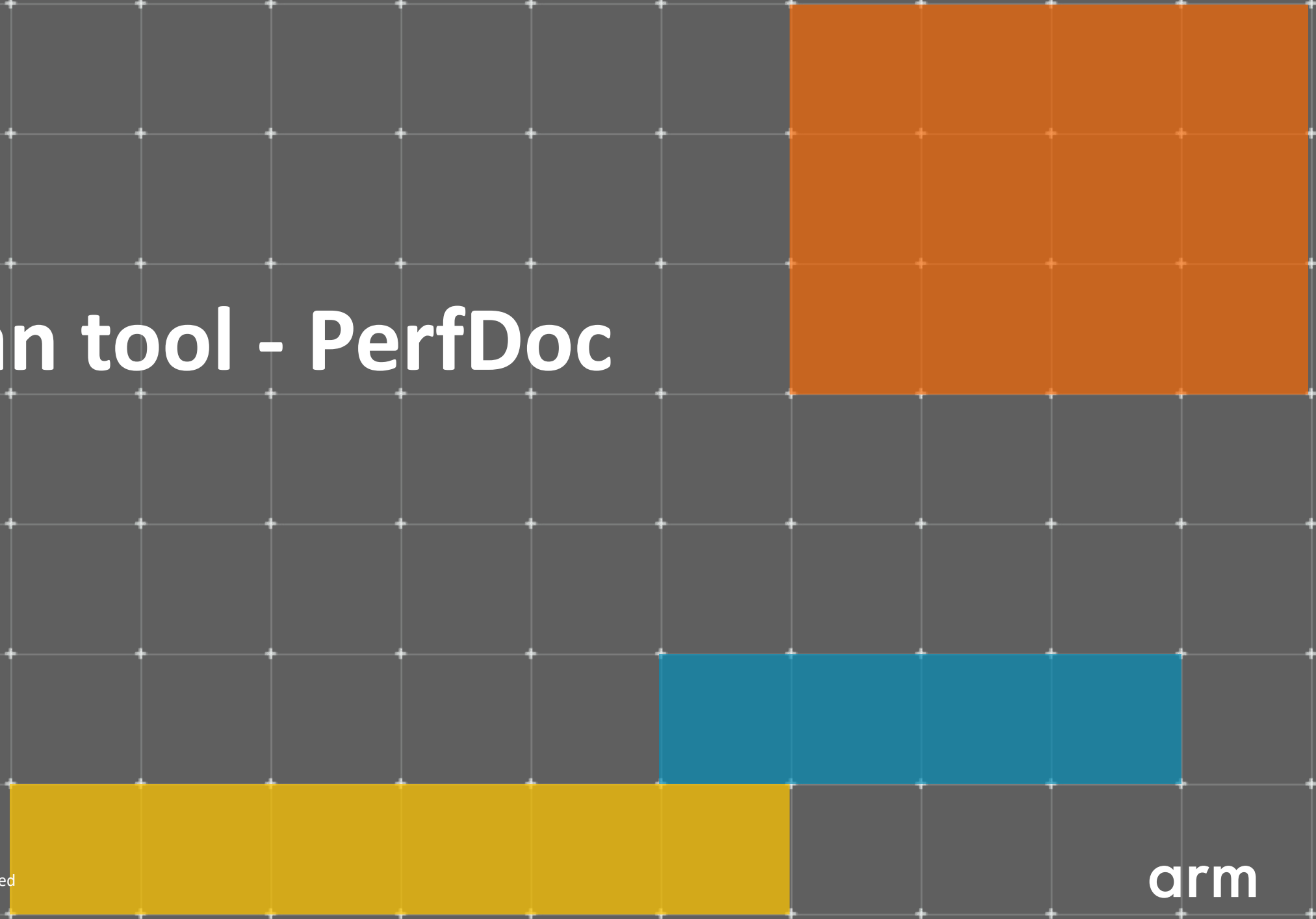
arm

Best practices

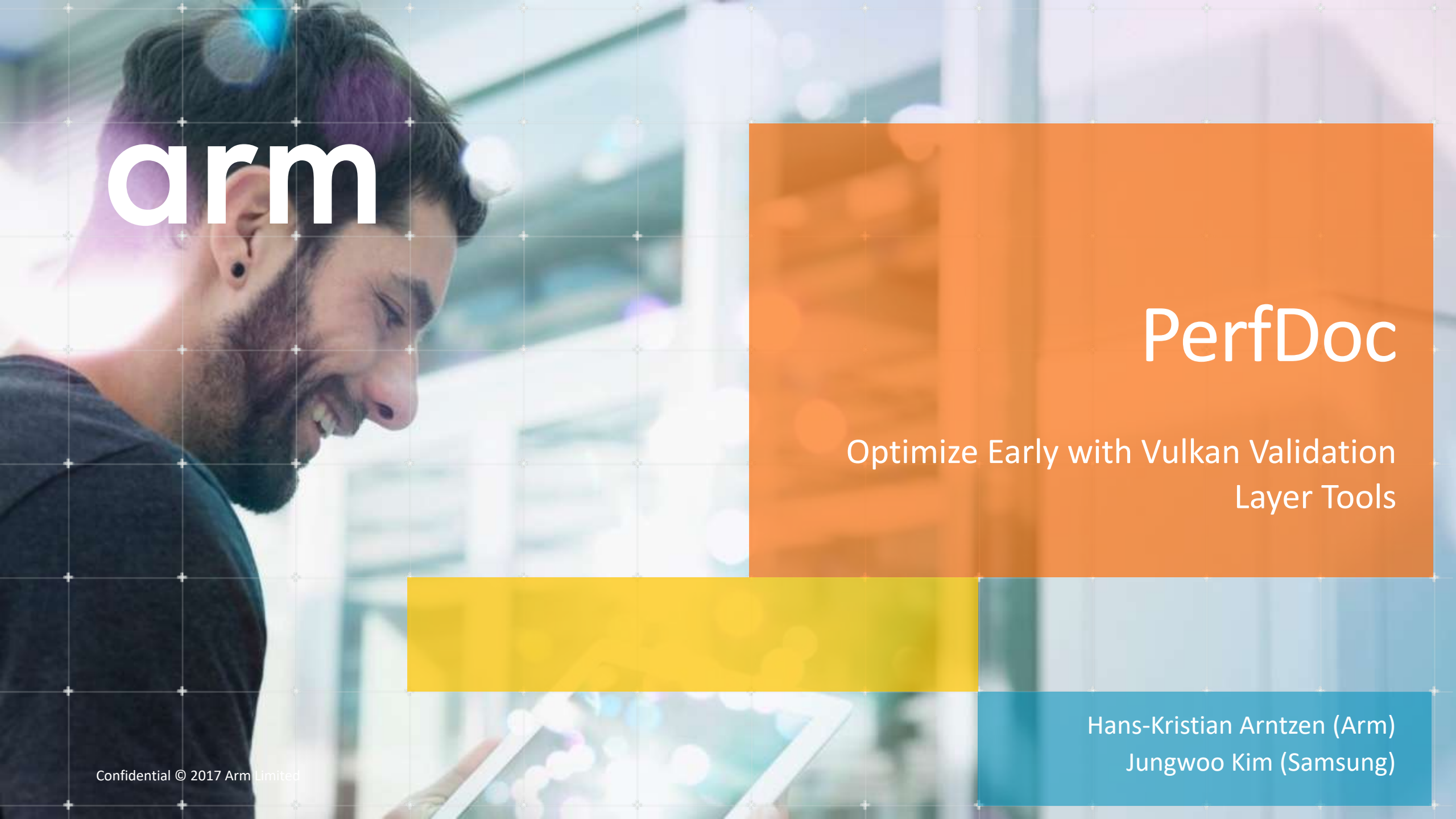
We released a PDF for expert developers which goes through a lot of best practices on Mali, both for Vulkan and GLES. https://developer.arm.com/docs/100971/latest

We are in the process of releasing an open source Vulkan layer which aims to automatically check your application for some of these errors.

Vulkan tool - PerfDoc



arm



PerfDoc

Optimize Early with Vulkan Validation Layer Tools

Hans-Kristian Arntzen (Arm)
Jungwoo Kim (Samsung)

What PerfDoc would help

- Can be used in Linux/Windows and Android.
- PerfDoc is to be used during development to catch potential performance issues early.
- The layer will run on any Vulkan implementation, so Mali-related optimizations can be found even when doing bringup on desktop platforms
- Errors are reported either through VK_EXT_debug_report to the application as callbacks, or via console/logcat if enabled

Explicit APIs have performance pitfalls

- With Vulkan, more control is placed in the hands of developers.
- More control means less, or no handholding from driver.
- If you get performance wrong, you can get it very wrong.
- Hardware vendors need to step up to make this easier for developers.

A best practices document

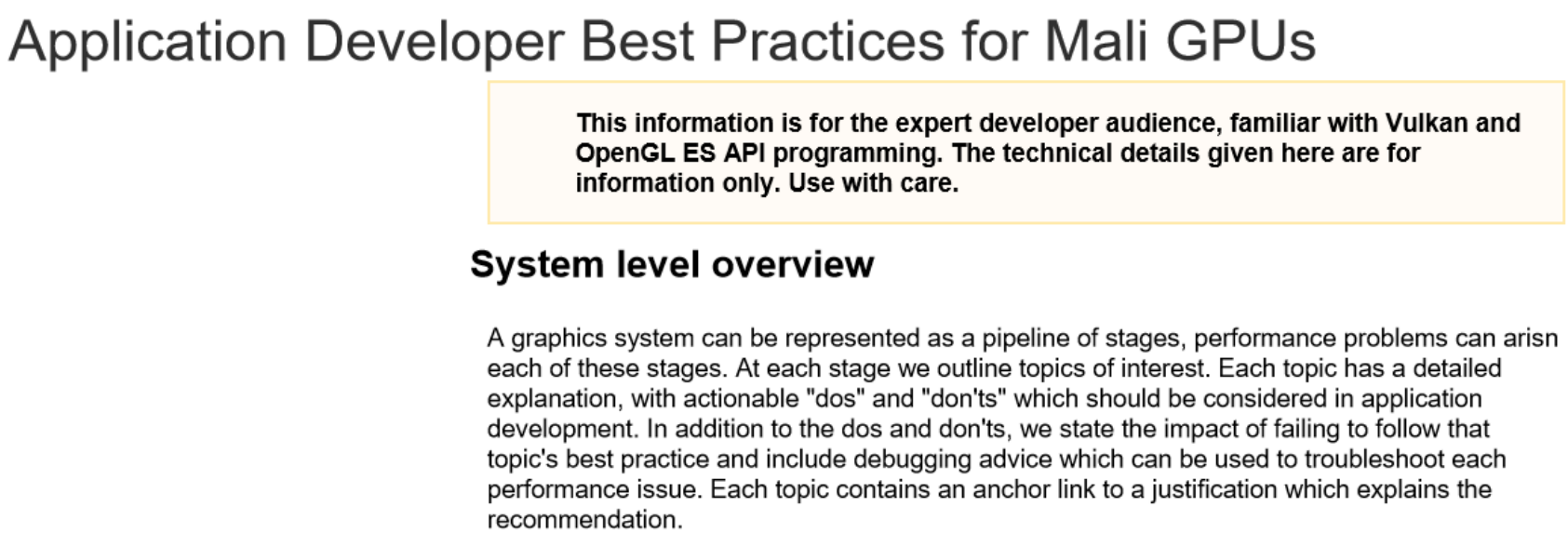
Guiding you through the initial design process

Before starting to design your backend

Cannot be validated automatically

Mistakes can and will be made in larger systems

Hardware vendors can **rarely** audit engine source code



Application Developer Best Practices for Mali GPUs

This information is for the expert developer audience, familiar with Vulkan and OpenGL ES API programming. The technical details given here are for information only. Use with care.

System level overview

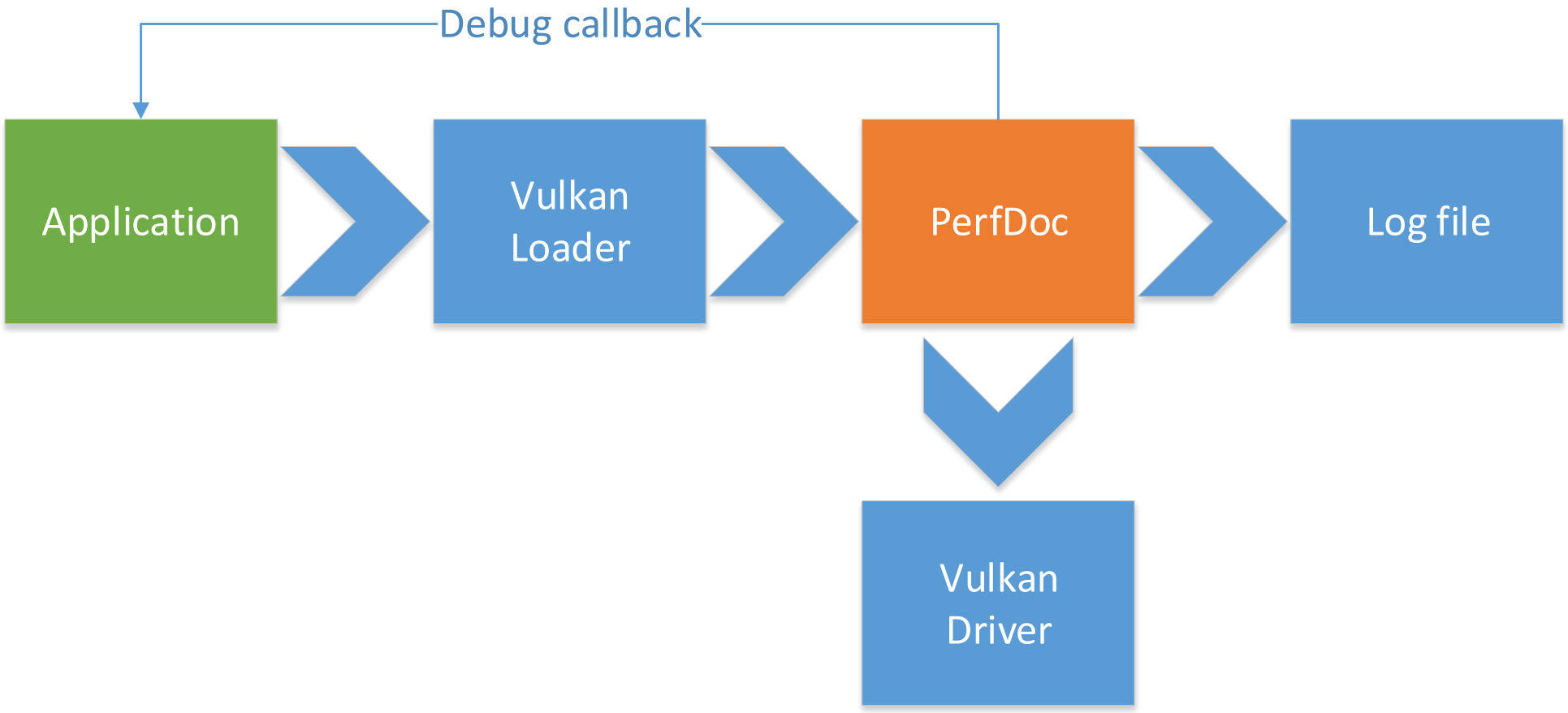
A graphics system can be represented as a pipeline of stages, performance problems can arisn each of these stages. At each stage we outline topics of interest. Each topic has a detailed explanation, with actionable "dos" and "don'ts" which should be considered in application development. In addition to the dos and don'ts, we state the impact of failing to follow that topic's best practice and include debugging advice which can be used to troubleshoot each performance issue. Each topic contains an anchor link to a justification which explains the recommendation.

arm

Programmatically checking best practices

- It is very labor intensive to tell if a best practice for a GPU is broken or not without automated tools.
- In an ideal world, this document could automatically check your app for all pitfalls when you run it.
- We have the validation layers for similar reasons. The Vulkan specification cannot check your app automatically.

System overview



arm

Links

PerfDoc:

- https://github.com/ARM-software/PerfDoc

Arm Mali application developer best practices

- https://developer.arm.com/docs/100971/latest/arm-mali-application-developer-best-practices-developer-guide

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद