Moving to Vulkan:
How to make your 3D graphics more explicit

Introduction & Welcome

Alon Or-bach, Samsung Electronics

@alonorbach (disclaimers apply!)

Welcome!

- **Housekeeping announcement from our hosts at ARM**
- **Etiquette for questions & engaging our online participants**
- **Outline for the day**
- **Quick intro to Khronos and the Khronos UK Chapter**
- **Interact with us on Twitter**
  - @KhronosUK
  - #MovingToVulkan
- **Tell us what you think about the day**
  - and most importantly, what you'd like us to do that we didn't

Moving to Vulkan: Today's agenda

Approximate timings for the day	Event	Speaker
9:00 – 10:00 am	Registration, demos, Q&A clinics, networking and coffee on arrival	
10:00 – 10:15 am	Welcome and Khronos UK Chapter Intro	Alon Or-bach, Samsung Electronics
10:15 – 11:00 am	Vulkan 1-0-1	Tom Olson, ARM
11:00 – 11:30 am	Command buffers	Michael Worcester, Imagination
11:30 – 12:15 pm	SPIR-V and GLSL, SPIR-V Cross Tool	Neil Hickey, ARM Hans-Kristian Arntzen, ARM
12:15 – 1:15 pm	Lunch break and demos, Q&A clinics & networking	
1:15 – 1:45 pm	Vertex Fetch and resource descriptors	Jesse Barker, ARM
1:45 – 2:15 pm	Render passes	Andrew Garrard, Samsung Electronics
2:15 – 2:45 pm	Synchronisation	Tobias Hector, Imagination
2:45 – 3:00 pm	Coffee break, demos, Q&A clinics and networking	
3:00 – 3:30 pm	Swapchains	Alon Or-bach, Samsung
3:30 – 4:00 pm	Simultaneous Graphics & Compute	Chris Hebert, NVIDIA
4:00 – 4:30 pm	Porting apps to Vulkan	Hans-Kristian Arntzen, ARM
4:30 – 5:30 pm	Panel discussion – Moving to Vulkan: Lessons to note when going explicit	
5:30 pm	Leaving by coach to the Cambridge Beer Festival to network further	

BOARD OF PROMOTERS

KHRONOS GROUP

Over 100 members worldwide
any company is welcome to join

Khronos Connects Software to Silicon

Industry Consortium creating OPEN STANDARD APIs for hardware acceleration
Any company is welcome – one company one vote

Software

ROYALTY-FREE specifications
State-of-the art IP framework protects members AND the standards

Conformance Tests and Adopters Programs for specification integrity and cross-vendor portability

Silicon

Low-level silicon APIs needed on almost every platform: graphics, parallel compute, rich media, vision, sensor and camera processing

International, non-profit organization
Membership and Adopters fees cover operating and engineering expenses

Strong industry momentum
100s of man years invested by industry experts

Well over a *BILLION* people use Khronos APIs *Every Day*…

KHRONOS GROUP

What is a Khronos Chapter?

- **Geographical group of people keen to talk technology**
- **Encourage adoption of Khronos standards**
  - Get the word out on the latest developments in APIs
  - Share experience of using Khronos APIs and related tech
- **Get feedback on how features are being used, offer advice**
- **Gather developer community requirements back into Khronos**

Vulkan 101

Tom Olson
Directory, Graphics Research, ARM
Chair, Vulkan Working Group

What is Vulkan?

- **A 3D graphics API for the next 20 years**
  - Logical successor to OpenGL / OpenGL ES
  - Modern, efficient design
  - An open, industry-controlled standard

- **Here, now**
  - Released in February 2016
  - Available today for Windows / Linux
  - Shipping in Samsung Galaxy S7
  - Support announced in Android ‘N’

- **Different!**
  - Fundamental change in philosophy
  - Requires corresponding changes in applications

Why did we do this?

- Traditional APIs had issues…
- Developers weren't happy

http://www.joshbarczak.com/blog/?p=154

http://richg42.blogspot.com/2014/05/things-that-drive-me-nuts-about-opengl.html

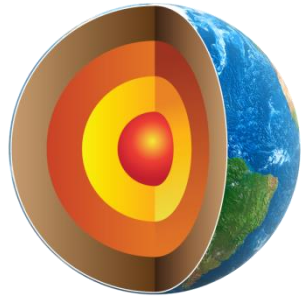
Problems with OpenGL / OpenGL ES

- **Programming model doesn't match GPU HW**
  - Especially in mobile
  - Driver magic hides the mismatch
- **CPU intensive**
  - Lots of state validation, dependency tracking
- **Complex, buggy, unpredictable drivers**
  - Different bugs and fast-paths on every GPU
- **Fundamentally single-threaded**
  - Can't use multi-core CPUs effectively
- **…not to mention twenty years of legacy cruft**

Enter Vulkan…

- **Design discussions start in October 2012**

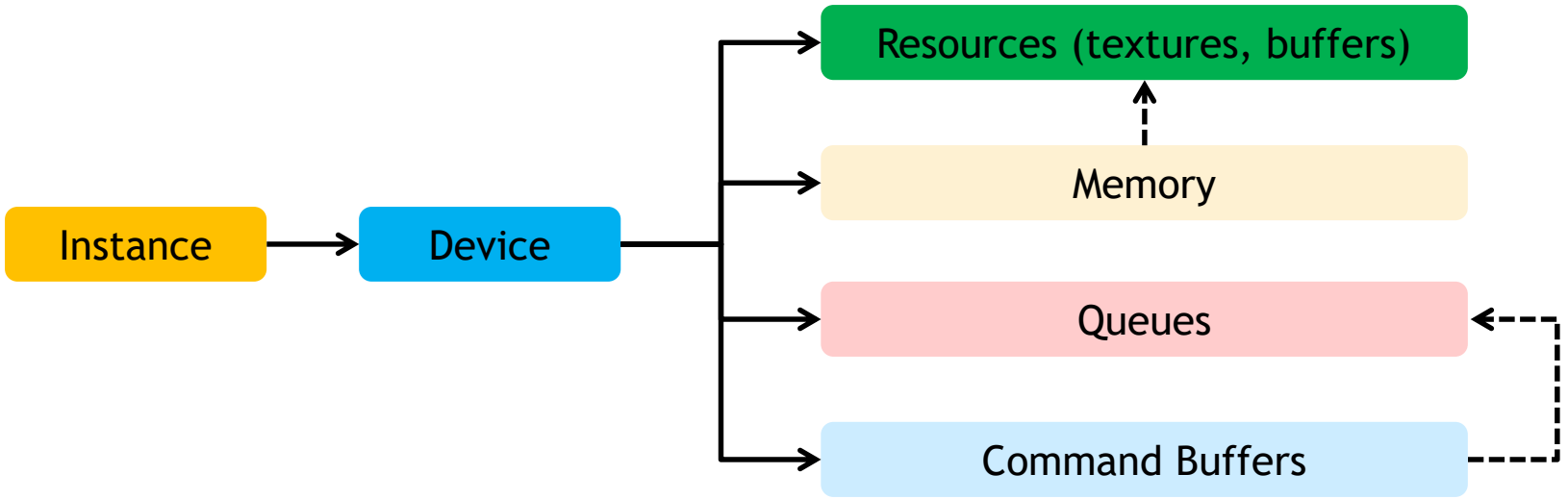
- **Moves into high gear in July/August 2014**
  - Commitment from key ISVs
  - AMD donation of Mantle



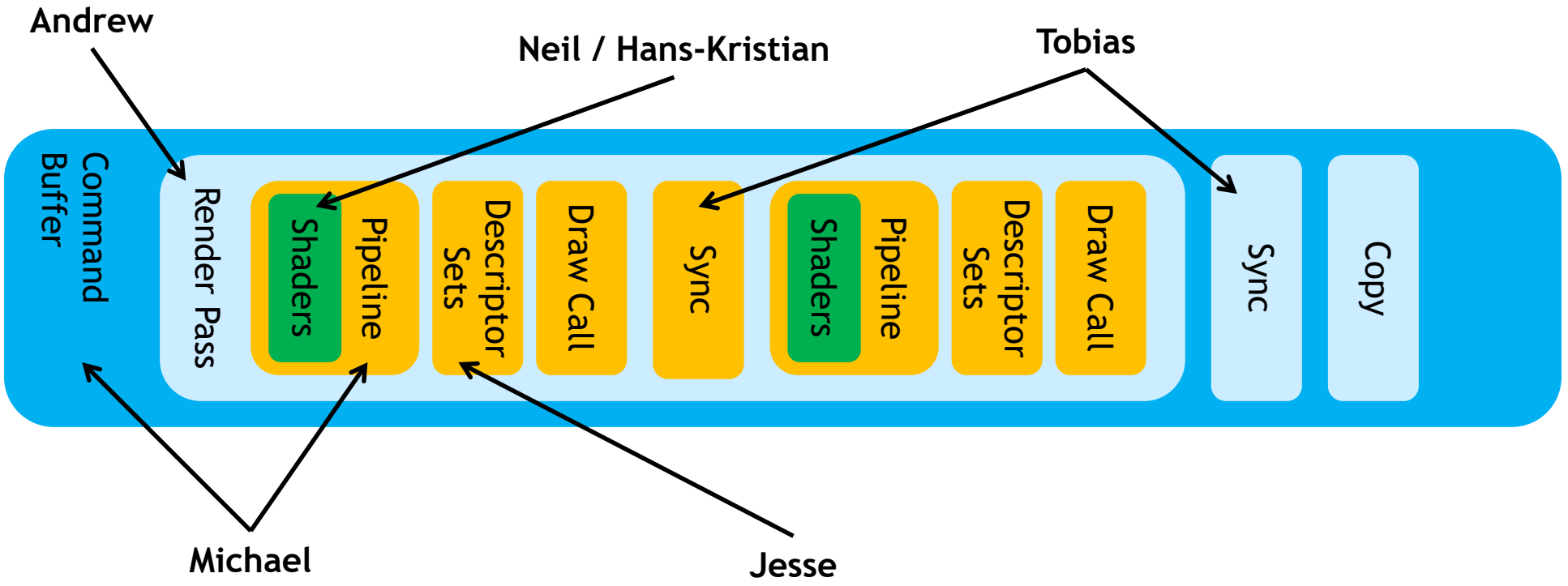
- **A lot of very hard work follows…**

- **Release to public in February 2016**
  - Conformant drivers from four IHVs
  - GLSL to SPIR-V compiler
  - Debug and validation tools

Vulkan in one slide



Vulkan in ~~one slide~~ two slides



The principle of *Explicit Control*

- **You promise to tell the driver**
  - *What* you are going to do
  - *In sufficient detail* that it doesn't have to guess
  - *When* the driver needs to know it

OpenGL lets you specify important information very late, and change it at any time. It's convenient, but has huge performance costs.

- **In return, driver promises to do**
  - *What* you asked for
  - *When* you asked for it
  - *Very quickly*

OpenGL drivers often defer work until later, move it to another thread, or even ignore your commands, based on guesses about your intent. Vulkan drivers won't.

- ***No driver magic!***

Loader, layers, and extensions

- **Vulkan has no dependencies on external APIs**
  - ICD loader is built-in
  - Window system binding is (semi) built-in

- **A side benefit: Layers**
  - Loader can install intercept libraries (“layers”)
  - E.g. trace, debug

- **Extensions**
  - Must be enabled at initialization time

Multithreading

- **All objects visible / accessible to all threads**

- **Most operations are *externally synchronized***
  - Application must prevent unsafe concurrent access
    - E.g., recording to the same command buffer
    - E.g., submitting to the same queue
  - Application must manage object lifetimes
  - Note, many objects are immutable
  - Concurrent read access is OK

- **Allocation / creation *are* internally synchronized and may block**
  - Per-thread pool allocators keep this reasonably cheap

Error handling

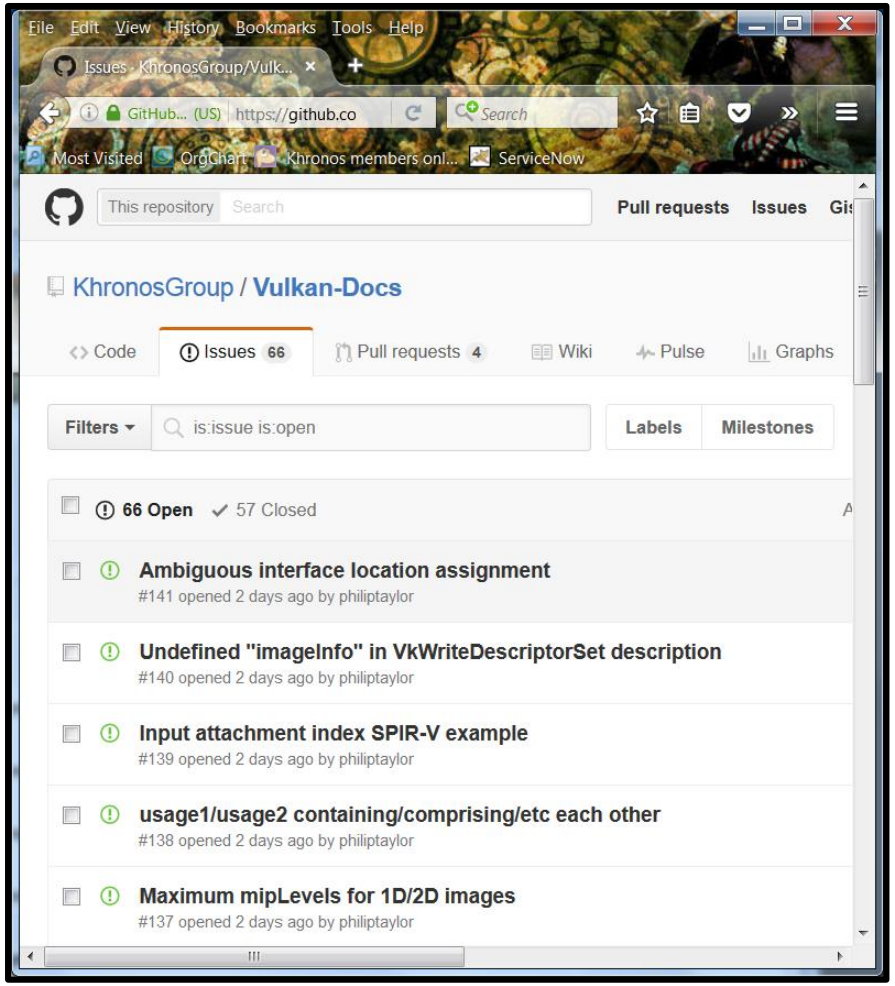
- **Vulkan is optimized for correct applications**
  - Does not (generally) check for invalid usage
  - Does not track dependencies
  - Does not (generally) provide thread safety
  - Breaking the rules results in undefined behavior

- **Vulkan *does* check for errors you can't predict**
  - Out of memory
  - Device lost
  - Other system errors…

- **Layers to the rescue!**
  - Can enable *validation layers* during development

Community

- **A new attitude**
  - ISV member input drove key decisions
  - Consulted with hundreds of developers
- **Strong commitment to open source**
  - Loader
  - Validation and other layers
  - SPIR-V tools: compiler, validator, …
  - Conformance tests
  - Specification
- **All at https://github.com/KhronosGroup**



Should you be using Vulkan?

- **Challenges**
  - Verbose and complex
  - Lots of exposed sharp edges
  - Lots to learn

- **Opportunities**
  - Much lower driver overhead
  - …which you can spread across multiple threads
  - More predictable performance
  - Mobile friendly

- **Realities**
  - Ecosystem is still immature
  - Will need to ship GL/DX versions for years to come

KHRONOS GROUP

Command Buffers and Pipelines

Michael Worcester – Driver Engineer
(michael.worcester@imgtec.com)
26 May 2016

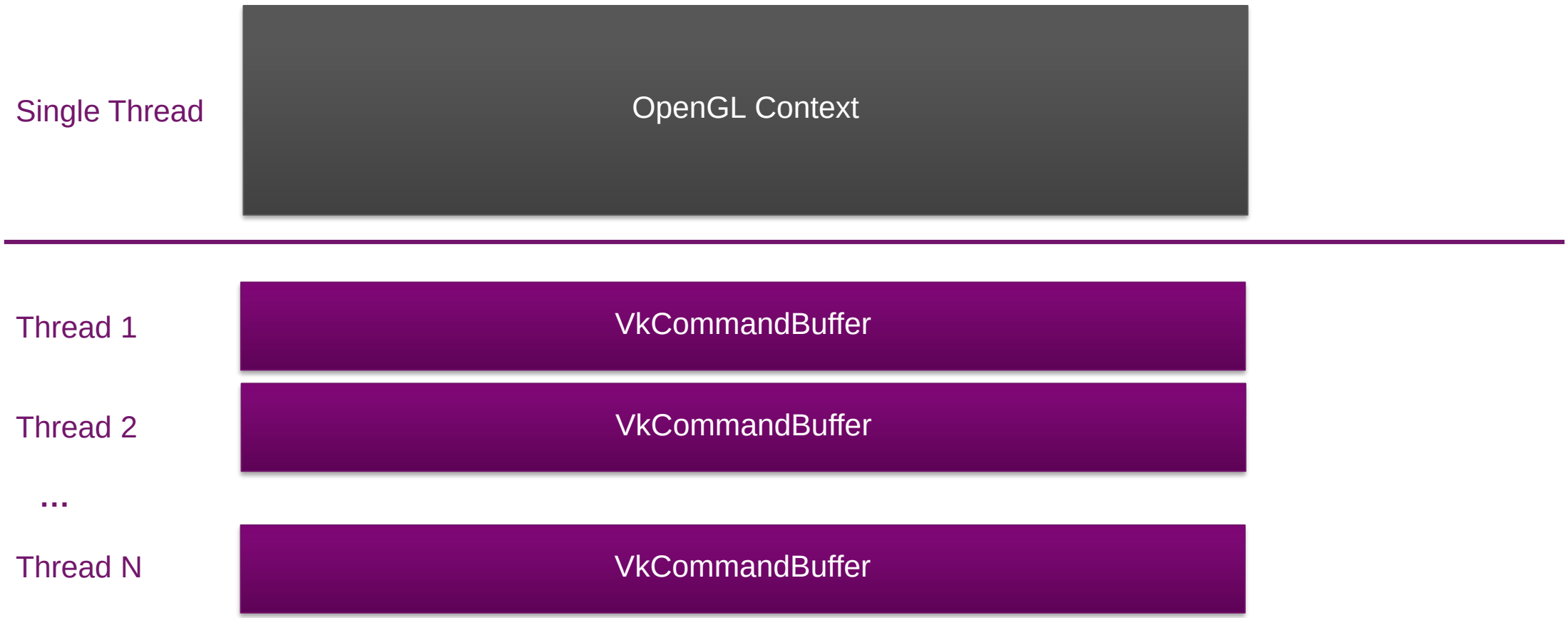
Command Buffers – Deferring the work

- **OpenGL is immediate (ignoring display lists)**
  - Driver does not know how much work is incoming
  - Has to guess
  - Bad!
- **Vulkan splits recording of work from submission of work**
  - Removes guesswork from driver
  - Reducing hitching
  - Helps eliminate unexplained resource usage

Command Buffers – Pooling Resource

- **Command Buffers always belong to a Command Pool**
  - Buffers are allocated from pools
  - Pools provide lightweight synchronisation
  - Pools can be reset, reclaiming all resources
  - Two flavours of pool:
    - Individual reset of command buffers
    - Group reset **only**

Command Buffers – Going wide



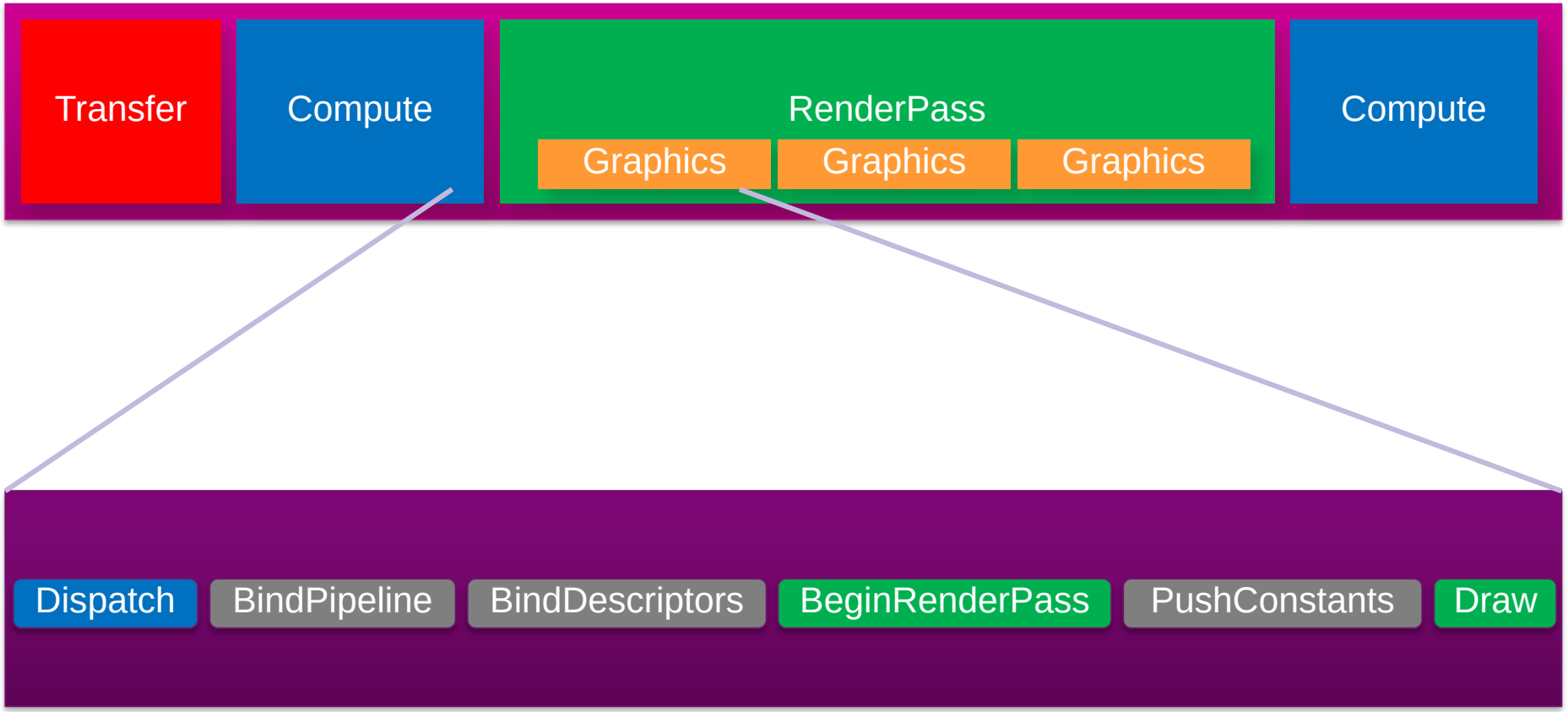
Command Buffers – Command Types

- **Deferred recording of commands**
  - Transfer
  - Graphics
  - Compute
  - Synchronisation

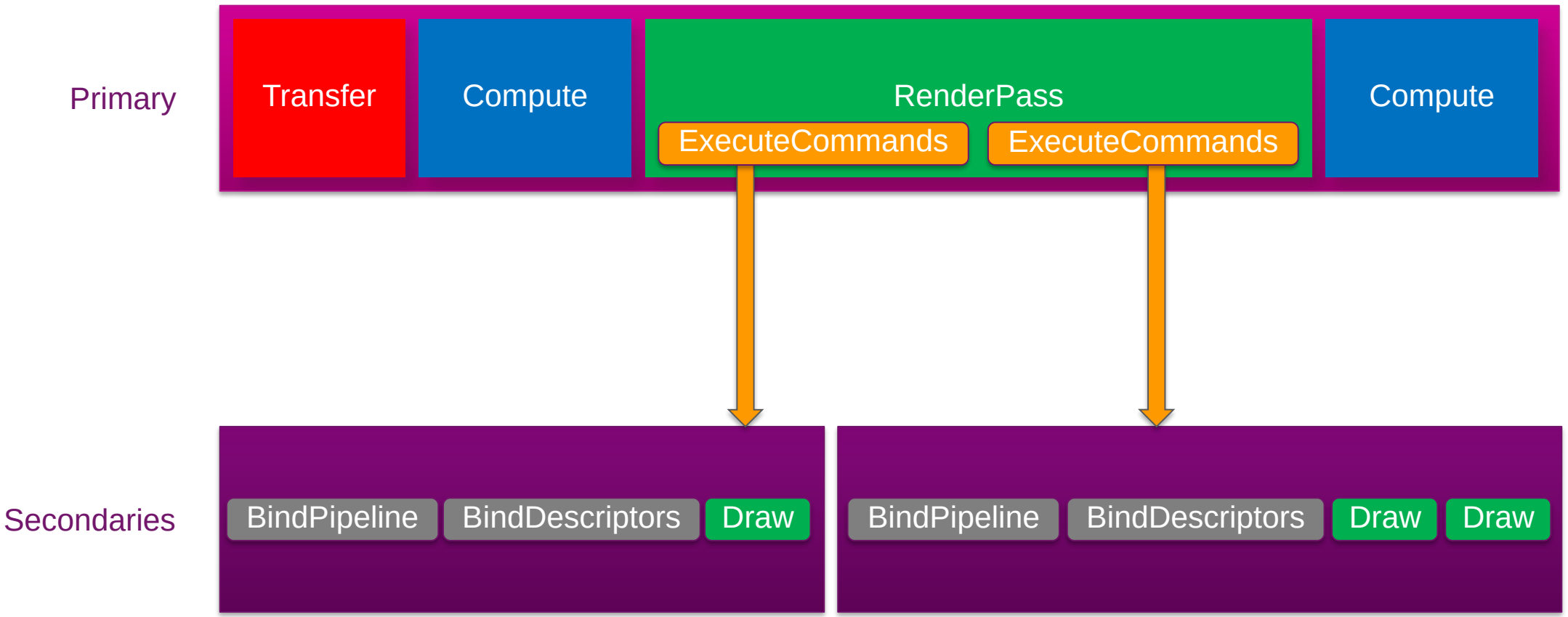
Command Buffers – Transfers

- **Transfer commands are raw copies**
  - However, they can change the *tiling* of an image (this is the only way!)
- **CPU -> GPU**
  - Texture upload
  - Static buffer data
- **GPU -> CPU**
  - Read back of data
- **GPU -> GPU**
  - Pipelined updates of data
  - Mipgen

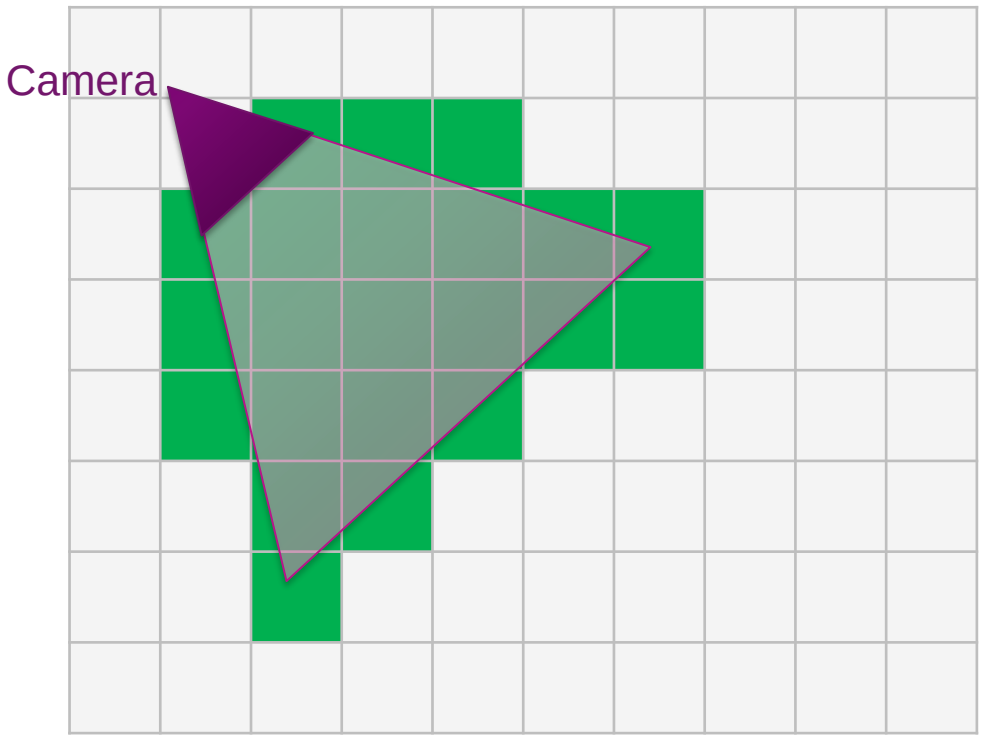
Command Buffers – “Inside” or “Out”



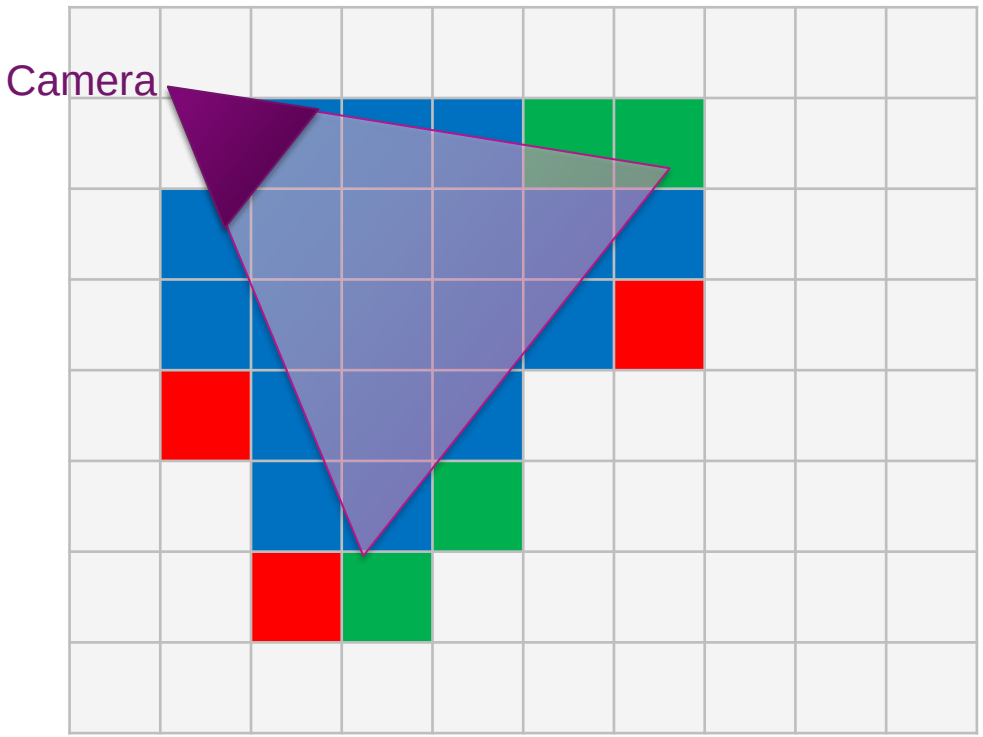
Command Buffers – Secondaries



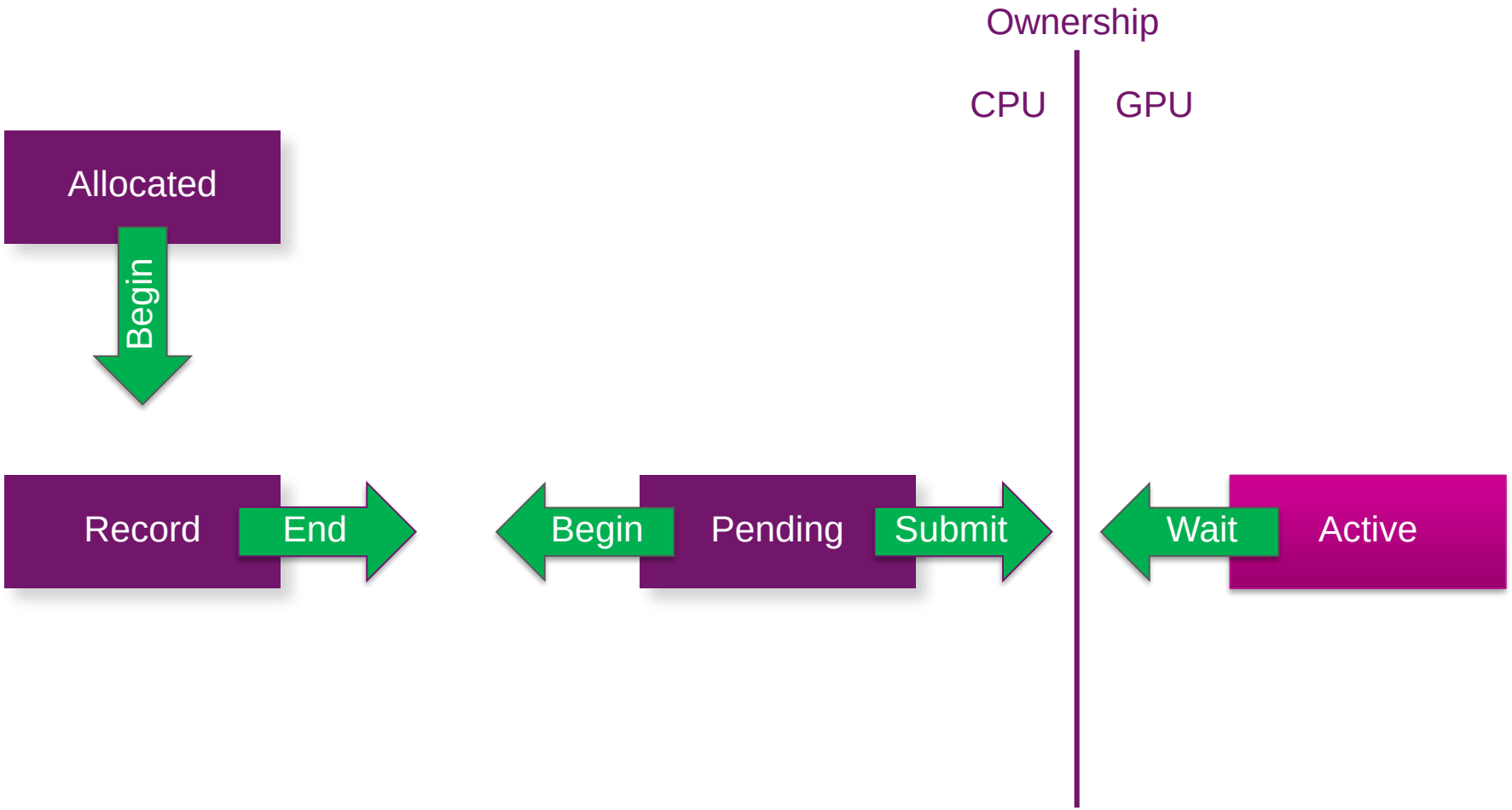
Command Buffers – Reuse



Command Buffers – Reuse



Command Buffers – Lifetime



Pipelines - An anatomy

- Fixed Function States
- Programmable Shaders
- Descriptor Layout
- Renderpass (more later)
- Dynamic State

Pipelines – Fixed Function States

- Everything that isn't a shader
- Buffer formats/layouts

- VertexInput
- InputAssembly
- Tessellation
- Viewport
- Raster
- Multisample
- DepthStencil
- ColorBlend

Pipelines – Shader Stages

- **Currently same as OpenGL**
  - Vertex
  - Control
  - Evaluation
  - Geometry
  - Fragment
- **Note: Tessellation and Geometry are optional features**

Pipelines – Descriptor Layout

Describes the set of resources that a shader can access

- Uniforms
- Storage Buffers
- Images
- Samplers
- Push Constants

Pipelines – Dynamic State

- **Per-draw state**
- **Tedious to compile each one**
  - Combinatorial explosion
- **Dynamic state!**
  - Opt-in
  - Only use when required

- **Viewport**
- **Scissor**
- **Line Width**
- **Depth Bias**
- **Blend Constant Colour**
- **Depth Bounds**
- **Stencil**
  - **Compare**
  - **Write**
  - **Reference**

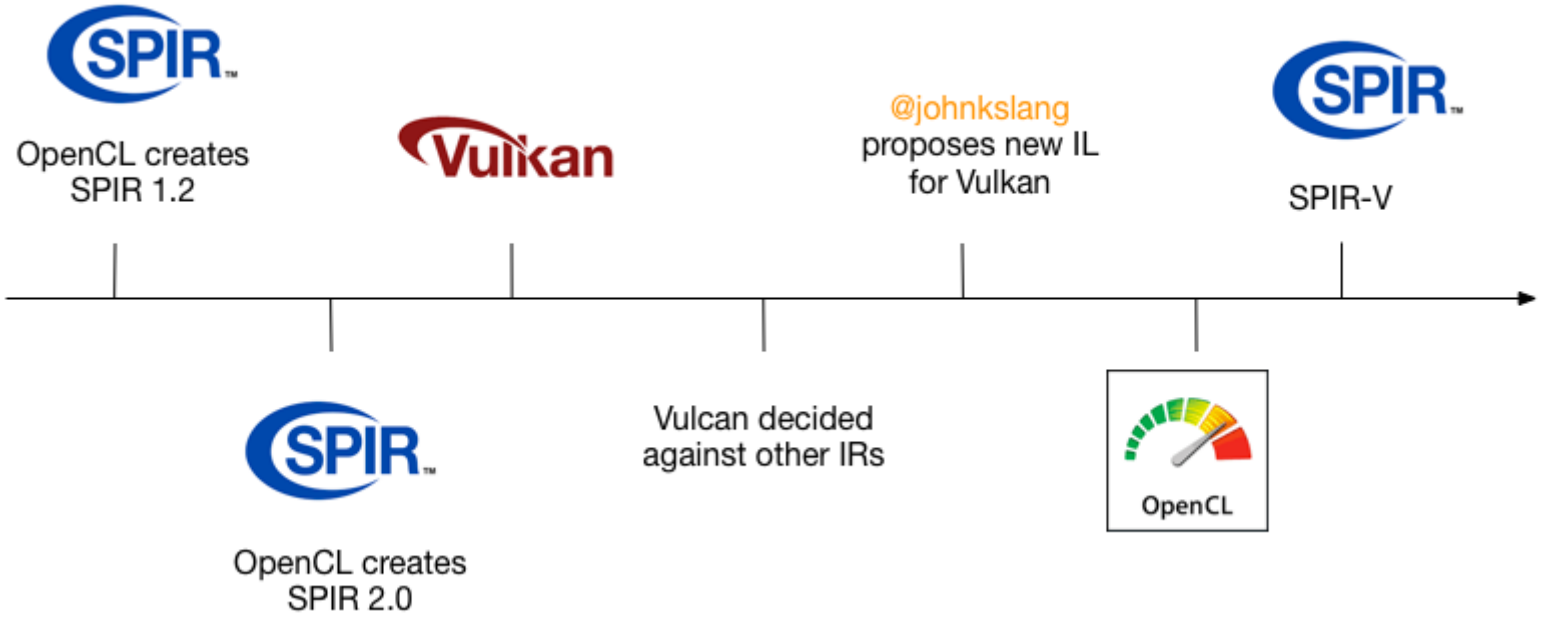
Pipelines – The Cache

- Share common state
- Load/Store

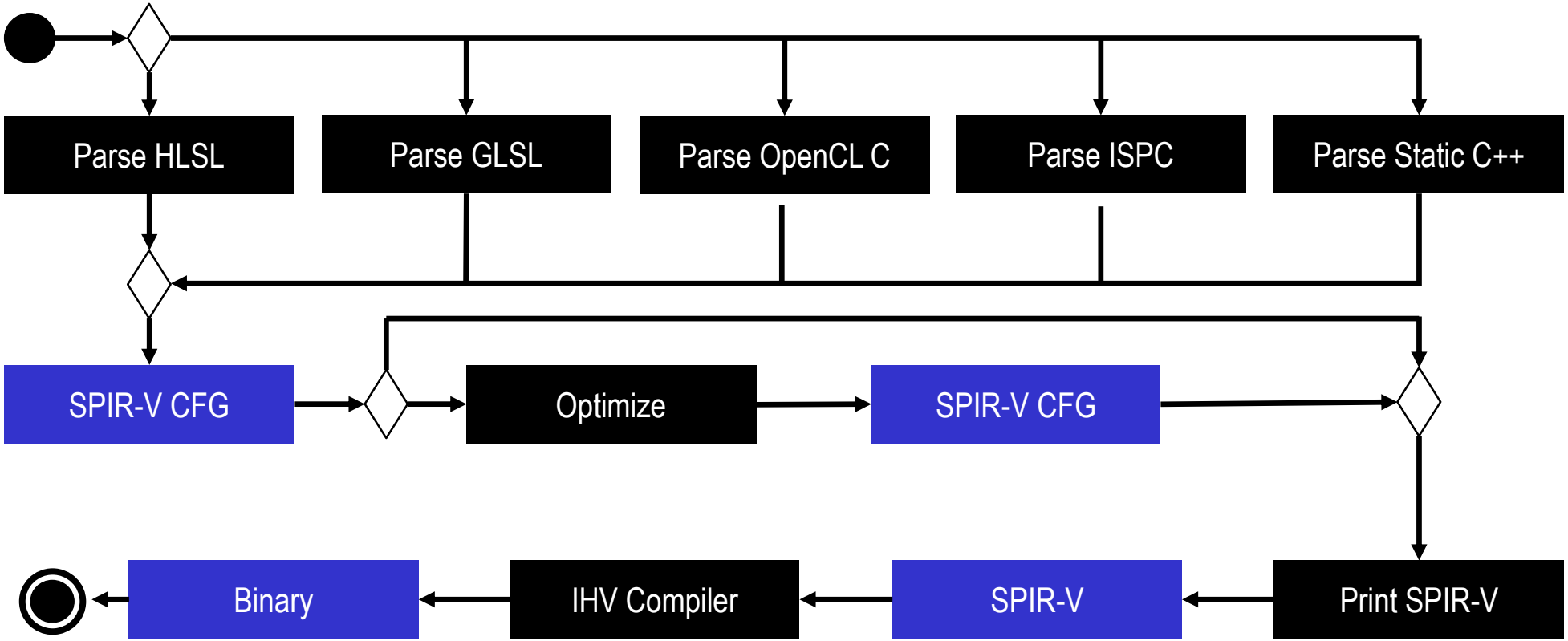
Introduction to SPIR-V Shaders

Neil Hickey
Compiler Engineer, ARM

SPIR History



SPIR-V Purpose



Developer Ecosystem

- Multiple Developer Advantages:
  - Same front-end compiler for multiple platforms
  - Reduces runtime kernel compilation time
  - Don't have to ship shader/kernel source code
  - Drivers are simpler and more reliable

Vulkan and OpenCL

	SPIR 1.2	SPIR 2.0	SPIR-V 1.0
LLVM Interaction	Uses LLVM 3.2	Uses LLVM 3.4	100% Khronos defined Round-trip lossless conversion
Compute Constructs	Metadata/Intrinsics	Metadata/Intrinsics	Native
Graphics Constructs	No	No	Native
Supported Language Feature Sets	OpenCL C 1.2	OpenCL C 1.2 OpenCL C 2.0	OpenCL C 1.2 – 2.0 OpenCL C++ and GLSL
OpenCL Ingestion	OpenCL C 1.2 Extension	OpenCL C 2.0 Extension	OpenCL 2.1 Core OpenCL 1.2 / 2.0 Extensions
Vulkan Ingestion	-	-	Vulkan 1.0 Core

KHRONOS GROUP

Compiler flow

Khronos has open sourced these tools and translators

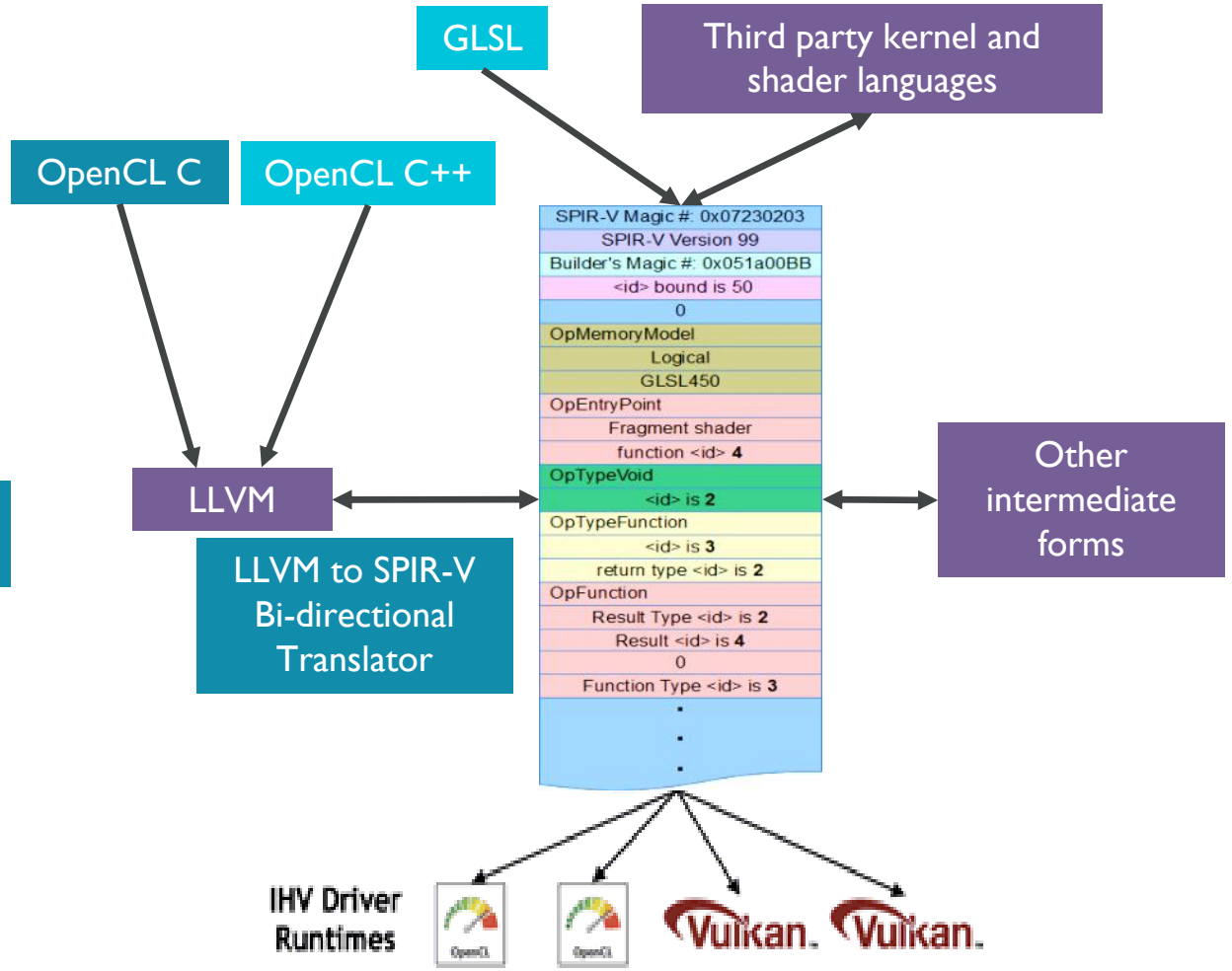
Khronos plans to open source these tools soon

SPIR-V Tools

SPIR-V Validator

SPIR-V (Dis)Assembler

SPIR-V
- 32-bit word stream
- Extensible and easily parsed
- Retains data object and control flow information for effective code generation and translation



SPIR-V Capabilities

- OpenCL and Vulkan
- Capabilities define feature sets
- Separate capabilities for Vulkan shaders and OpenCL kernels
- Validation layer checks correct capabilities requested

OpCapability Addresses
OpCapability Linkage
OpCapability Kernel
OpCapability Vector16
OpCapability Int16

SPIR-V Extensions

- OpExtension
- New functionality
- New instructions
- New semantics

```
OpExtInstImport
“OpenCL.std”
```

Vulkan shaders vs. GL shaders

- Program GLSL/ESSL shaders in high level language
- Ship high level source with application
- Graphics drivers compile at runtime
- Each driver needs a full compilation tool chain

- Shaders in binary format
- Compile offline
- Ship intermediate language with application
- Graphics drivers “just” lower from IL
- Higher level compilation can be shared among vendors (provided by Khronos)

Vulkan shaders vs. GL shaders

```
#version 310 es
precision mediump float;
uniform sampler2D s;
in vec2 texcoord;
out vec4 color;

void main()
{
        color = texture(s, texcoord);
}
```

```
; SPIR-V
; Version: 1.0
; Generator: Khronos GIslang Reference Front End; 1
; Bound: 20
; Schema: 0
             OpCapability Shader
        %1 = OpExtInstImport "GLSL.std.450"
             OpMemoryModel Logical GLSL450
             OpEntryPoint Fragment %4 "main" %9 %17
             OpExecutionMode %4 OriginUpperLeft
             OpSource ESSL 310
             OpName %4 "main"
             OpName %9 "color"
             OpName %13 "s"
             OpName %17 "texcoord"
             OpDecorate %9 RelaxedPrecision
             OpDecorate %13 RelaxedPrecision
             OpDecorate %13 DescriptorSet 0
             OpDecorate %14 RelaxedPrecision
             OpDecorate %17 RelaxedPrecision
             OpDecorate %18 RelaxedPrecision
             OpDecorate %19 RelaxedPrecision
        %2 = OpTypeVoid
        %3 = OpTypeFunction %2
```

```
%6 = OpTypeFloat 32
        %7 = OpTypeVector %6 4
        %8 = OpTypePointer Output %7
        %9 = OpVariable %8 Output
       %10 = OpTypeImage %6 2D 0 0 0 1 Unknown
       %11 = OpTypeSampledImage %10
       %12 = OpTypePointer UniformConstant %11
       %13 = OpVariable %12 UniformConstant
       %15 = OpTypeVector %6 2
       %16 = OpTypePointer Input %15
       %17 = OpVariable %16 Input
        %4 = OpFunction %2 None %3
        %5 = OpLabel
       %14 = OpLoad %11 %13
       %18 = OpLoad %15 %17
       %19 = OpImageSampleImplicitLod %7 %14 %18
             OpStore %9 %19
             OpReturn
             OpFunctionEnd
```

Khronos SPIR-V Tools

- Reference frontend (glslang)

```
glslangValidator –V –o shader.spv shader.frag
```

- SPIR-V disassembler (spirv-dis)

```
spirv-dis -o shader.spvasm shader.spv
```

- SPIR-V assembler (spirv-as)

```
spirv-as –o shader.spv shader.spvasm
```

- SPIR-V reflection (spirv-cross)

```
spirv-cross shader.spv
```

Vulkan shaders in a high level language

- GL_KHR_vulkan_glsl
- Exposes SPIR-V features
- Similar to GLSL with some changes
- Extends #version 140 and higher on desktop and #version 310 es for mobile content

KHRONOS GROUP

Vulkan_glsl removed features

- Default uniforms
- Atomic-counter bindings
- Subroutines
- Packed block layouts

Vulkan_glsl new features

- Push constants
- Separate textures and samplers
- Descriptor sets
- Specialization constants
- Subpass inputs

Push Constants

- **Push constants replace non-opaque uniforms**
  - Think of them as small, fast-access uniform buffer memory
- **Update in Vulkan with vkCmdPushConstants**

```
// New
layout(push_constant, std430) uniform PushConstants {
    mat4 MVP;
    vec4 MaterialData;
} RegisterMapped;

// Old, no longer supported in Vulkan GLSL
uniform mat4 MVP;
uniform vec4 MaterialData;

// Opaque uniform, still supported
uniform sampler2D sTexture;1
```

Separate textures and samplers

- sampler contains just filtering information
- texture contains just image information
- combined in code at the point of texture lookup

```
uniform sampler s;
uniform texture2D t;
in vec2 texcoord;
...
void main()
{
        fragColor = texture(sampler2D(t,s), texcoord);
}
```

Descriptor sets

- Bound objects can optionally define a descriptor set
- Allows bound objects to be updated in one block
- Allows objects in other descriptor sets to remain the same
- Enabled with the set = ... syntax in the layout specifier

```
layout(set = 0, binding = 0) uniform sampler s;
layout(set = 1, binding = 0) uniform texture2D t;
```

Specialization constants

- Allows for special constants to be created whose value is overridable at pipeline creation time.
- Can be used in expressions
- Can be combined with other constants to form new specialization constants
- Declared using layout(constant_id=...)
- Can have a default value if not overridden at runtime

```
layout(constant_id = 1) const int arraySize = 12;

vec4 data[arraySize];
```

Specialization constants(2)

- gl_WorkGroupSize can be specialized with values for the x,y and z component.

```
layout(local_size_x_id = 2, local_size_z_id = 3) in;
```

- These specialization constants can be set at pipeline creation time by using vkSpecializationMapInfo

```
const VkSpecializationMapEntry entries[] =
{
{  1,                  // constantID
   0*sizeof(uint32_t), // offset
   sizeof(uint32_t)    // size
},
};
```

Specialization constants(3)

```
const uint32_t data[] = { 16};
const VkSpecializationInfo info =
{
        1,         // mapEntryCount
   entries,  // pMapEntries
   1*sizeof(uint32_t),  // dataSize
        data,    // pData
};
```

Subpass Inputs

- Vulkan supports subpasses within render passes
- Standardized GL_EXT_shader_pixel_local_storage!

```
// GLSL
#extension GL_EXT_shader_pixel_local_storage : require
__pixel_local_inEXT GBuffer {
    layout(rgba8) vec4 albedo;
    layout(rgba8) vec4 normal;
    ...
} pls;

// Vulkan
layout(input_attachment_index = 0) uniform subpassInput albedo;
layout(input_attachment_index = 1) uniform subpassInput normal;
...
```

Acknowledgements

- Hans-Kristian Arntzen – ARM
- Benedict Gaster – University of the West of England
- Neil Henning – Codeplay

Using SPIR-V in practice with SPIRV-Cross

Hans-Kristian Arntzen
Engineer, ARM

Contents

- **Moving to offline compilation of SPIR-V**
- **Creating pipeline layouts with SPIRV-Cross**
  - Descriptor sets
  - Push constants
  - Multipass input attachments
- **Making SPIR-V portable to other graphics APIs**
- **Debugging complex shaders with your C++ debugger of choice**

KHRONOS GROUP

Offline Compilation to SPIR-V

- Shader compilation can be part of your build system
- Catching compilation bugs in build time is always a plus
- Strict, mature GLSL frontends available
  - glslang: https://github.com/KhronosGroup/glslang
  - shaderc: https://github.com/google/shaderc
- Full freedom for other languages in the future

```
# Makefile rules

FRAG_SHADERS := $(wildcard *.frag)
SPIRV_FILES :=
$(FRAG_SHADERS:.frag=.frag.spv)

shaders: $(SPIRV_FILES)

%.frag.spv: %.frag
    glslc –o $@ $< $(GLSL_FLAGS) –std=310es
```

Vulkan Pipeline Layouts

- Need to know the “function signature” of our shaders

```
pipelineInfo.layout = <layout goes here>;
vkCreateGraphicsPipelines(..., &pipelineInfo, ..., &pipeline);
```

The Contents of a Pipeline Layout

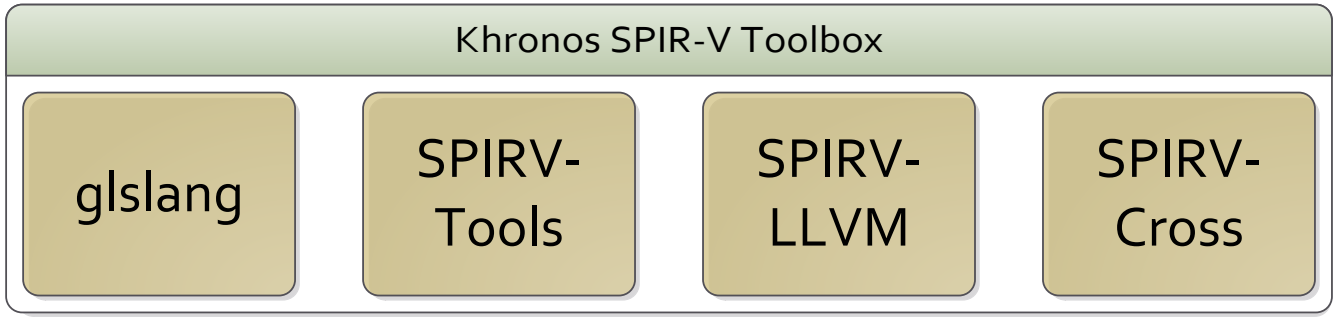

```
layout(set = 0, binding = 1) uniform UBO {
          mat4 MVP;
};
layout(set = 1, binding = 2) uniform sampler2D uTexture;
layout(push_constant) uniform PushConstants {
          vec4 FastConstant;
} constants;
```

- 16 bytes of push constant space
- Two descriptor sets
- Set #0 has one UBO at binding #1
- Set #1 has one combined image sampler at binding #2

- **Need to figure this out automatically, or write every layout by hand**
  - Latter is fine for tiny applications
  - Vulkan does not provide reflection here, after all, this is vendor neutral information

Introducing SPIRV-Cross

- SPIRV-Cross is a new tool hosted by Khronos
  - https://github.com/KhronosGroup/SPIRV-Cross
- Extensive reflection
- Decompilation to high level languages



Reflecting Uniforms and Samplers

- SPIRV-Cross has a simple API to retrieve resources

```
using namespace spirv_cross;

vector<uint32_t> spirv_binary = load_spirv_file();
Compiler comp(move(spirv_binary));

// The SPIR-V is now parsed, and we can perform reflection on it.
ShaderResources resources = comp.get_shader_resources();

for (auto &u : resources.uniform_buffers)
{
        uint32_t set = comp.get_decoration(u.id, spv::DecorationDescriptorSet);
        uint32_t binding = comp.get_decoration(u.id, spv::DecorationBinding);
        printf("Found UBO %s at set = %u, binding = %u!\n",
               u.name.c_str(), set, binding);
}
```

Stepping it up with Push Constants

- **SPIRV-Cross can figure out which push constant elements are in use**
  - Push constant blocks are typically shared across the various stages
  - Only parts of the push constant block are referenced in a single stage

```
layout(push_constant) uniform PushConstants {
          mat4 MVPInVertex;
          vec4 ColorInFragment;
} constants;

FragColor = constants.ColorInFragment; // Fragment only uses element #1.
```

```
uint32_t id = resources.push_constant_buffers[0].id;
vector<BufferRange> ranges = comp.get_active_buffer_ranges(id);
for (auto &range : ranges)
{
          printf("Accessing member #%u, offset %u, size %u\n",
                 range.index, range.offset, range.range);
}

// Possible to get names for struct members as well ☺
```

Subpass Input Attachments

- **Subpass attachments are similar to regular images**
  - Set
  - Binding
  - Input attachment index

```
layout(set = 0, binding = 0, input_attachment_index = 0) uniform subpassInput uAlbedo;
layout(set = 0, binding = 1, input_attachment_index = 1) uniform subpassInput uNormal;

vec4 lastColor = subpassLoad(uLastPass);
```

```
for (auto &attachment : resources.subpass_inputs)
{
        // ...
}
```

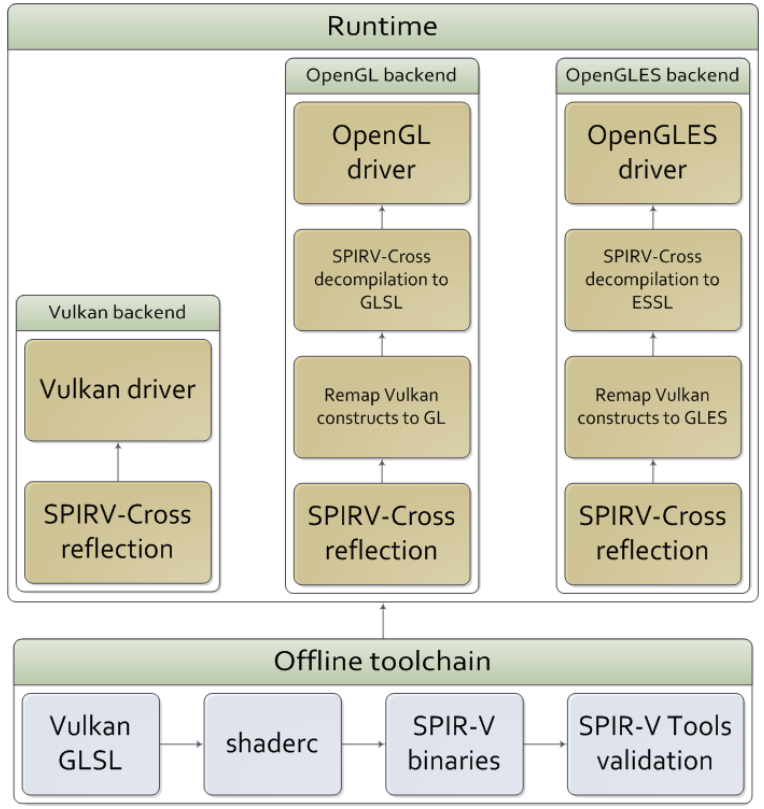
Taking SPIR-V Beyond Vulkan

- **SPIR-V is a great format to rally around**
  - Makes sense to be able to use it in older graphics APIs as well
- **Will take some time before exclusive Vulkan support is mainstream**
- **How to make use of Vulkan features while being compatible?**
  - Push constants
  - Subpass
  - Descriptor sets
- **Without tools, Vulkan features will be harder to take advantage of**

KHRONOS GROUP

GL + GLES + Vulkan Pipeline

- Implemented in our internal demo engine
- Write shaders in Vulkan GLSL
- Use Vulkan features directly
- No need for platform #ifdefs
- Can target mobile and desktop GL from same SPIR-V binary



Subpasses in OpenGL

- **The subpass attachment is really just a texture read from gl_FragCoord**
  - Enables reading directly from tile memory on tiled architectures
  - Great for deferred rendering and programmable blending

```
// Vulkan GLSL
uniform subpassInput uAlbedo;
...
FragColor = accumulateLight(
                              subpassLoad(uAlbedo),
                              subpassLoad(uNormal).xyz,
                              subpassLoad(uDepth).x);

// Translated to GLSL in SPIRV-Cross
uniform sampler2D uAlbedo;
...
FragColor = accumulateLight(
                              texelFetch(uAlbedo, ivec2(gl_FragCoord.xy), 0),
                              texelFetch(uNormal, ivec2(gl_FragCoord.xy), 0).xyz,
                              texelFetch(uDepth,  ivec2(gl_FragCoord.xy), 0).x);
```

KHRONOS GROUP

Push Constants in OpenGL

- **Push constants bundle up old-style uniforms into buffer blocks**
  - Translates directly to uniform structs
  - Use reflection to stamp out a list of glUniform() calls

```
// Vulkan GLSL
layout(push_constant) uniform PushConstants {
          vec4 Material;
} constants;

FragColor = constants.Material;

// Translated to GLSL in SPIRV-Cross
struct PushConstants {
          vec4 Material;
};
uniform PushConstants constants;

FragColor = constants.Material;
```

Descriptor Sets in OpenGL

- OpenGL has a binding space per type
- Find some remapping scheme that fits your application
- SPIRV-Cross can tweak bindings before decompiling to GLSL

```
// Vulkan GLSL
layout(set = 1, binding = 1) uniform sampler2D uTexture;

// SPIRV-Cross
uint32_t newBinding = 4;
glsl.set_decoration(texture.id, spv::DecorationBinding, newBinding);
glsl.unset_decoration(texture.id, spv::DecorationDescriptorSet);
string glslSource = glsl.compile();

// GLSL
layout(binding = 4) uniform sampler2D uTexture;
```

gl_InstanceIndex in OpenGL

- **Vulkan adds the base instance to the instance ID**
  - GL does not ☹
  - Workaround is to have GL backend pass in the base index as a uniform

```
// Vulkan GLSL
layout(set = 0, binding = 0) uniform UBO {
          mat4 MVPs[MAX_INSTANCES];
};

gl_Position = MVPs[gl_InstanceIndex] * Position;

// GLSL through SPIRV-Cross
layout(binding = 0) uniform UBO {
          mat4 MVPs[MAX_INSTANCES];
};
uniform int SPIRV_Cross_BaseInstance; // Supplied by application

gl_Position = MVPs[(gl_InstanceID + SPIRV_Cross_BaseInstance)] * Position;
```

KHRONOS GROUP

Debugging Shaders in C++

- **If you have thought …**
  - "I wish I could assert() in a compute shader"
  - "I wish I could instrument a shader with logging"
  - "I wish I could use clang address sanitizer to debug out-of-bounds access"
  - "I want to reproduce a shader bug outside the driver"
  - "I want to run regression tests when optimizing a shader"
  - "I want to step through a compute thread in <insert C++ debugger here>"
- **… the C++ backend in SPIRV-Cross could be interesting**
- **Still a very experimental feature**
- **Hope to expand this further in the future**

Basic Idea

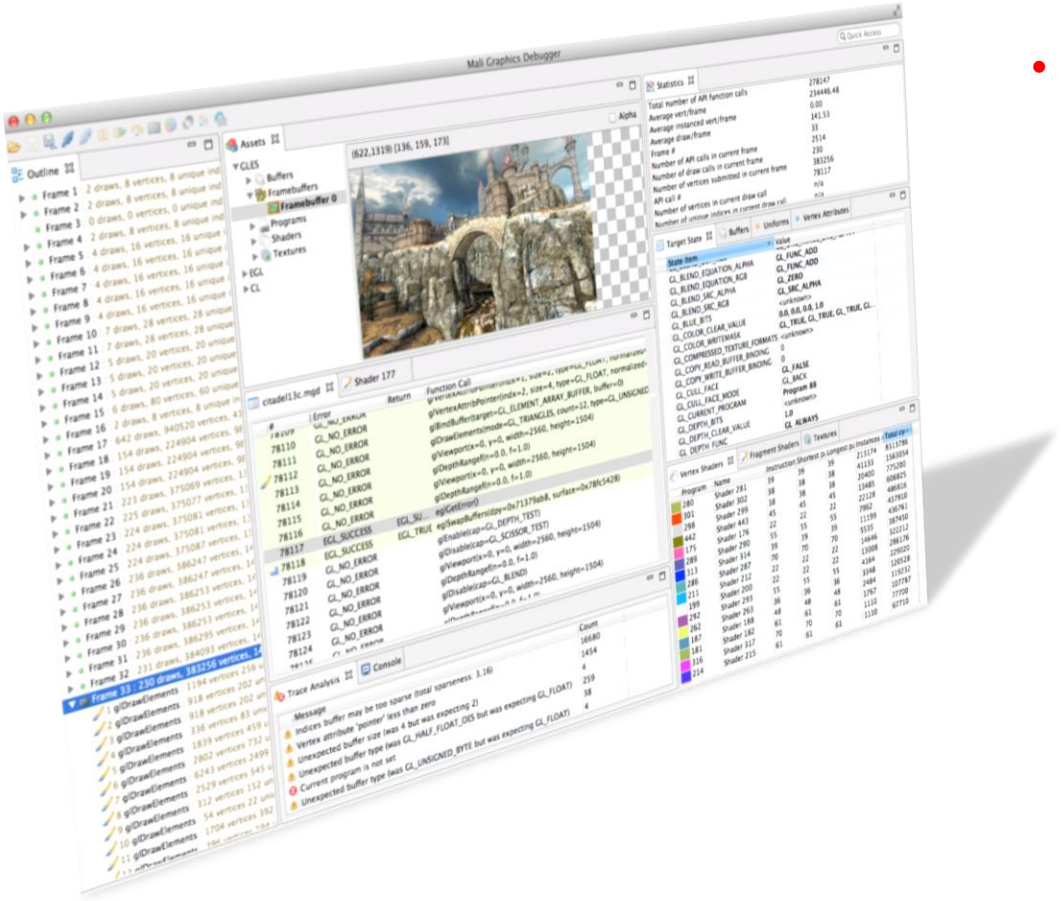
- **With GLM, C++ can be near GLSL compatible**
- **Reuse the GLSL backend to emit code which also works in C++**
  - Minor differences like references vs. in/out, etc
- **Add some scaffolding to redirect shader resources**
  - Easily done with macros, the actual C++ output is kept clean
- **The C++ output implements a simple C-compatible interface**
- **Add instrumentation to the C++ file as desired**
- **Compile C++ file to a dynamic library with debug symbols**
- **Instantiate from test program, bind buffers and invoke**
  - And have fun running shadertoy raymarchers at seconds per frame

On the Command Line


```
# Compile to SPIR-V
glslc –o test.spv test.comp

# Create C++ interface
spirv-cross --output test.cpp test.spv --cpp

# Add some instrumentation to the shader if you want
$EDITOR test.cpp

# Build library
g++ -o test.so –shared test.cpp –O0 –g –Iinclude/spirv_cross

# Run your test app
./<my app> --shader test.so
```

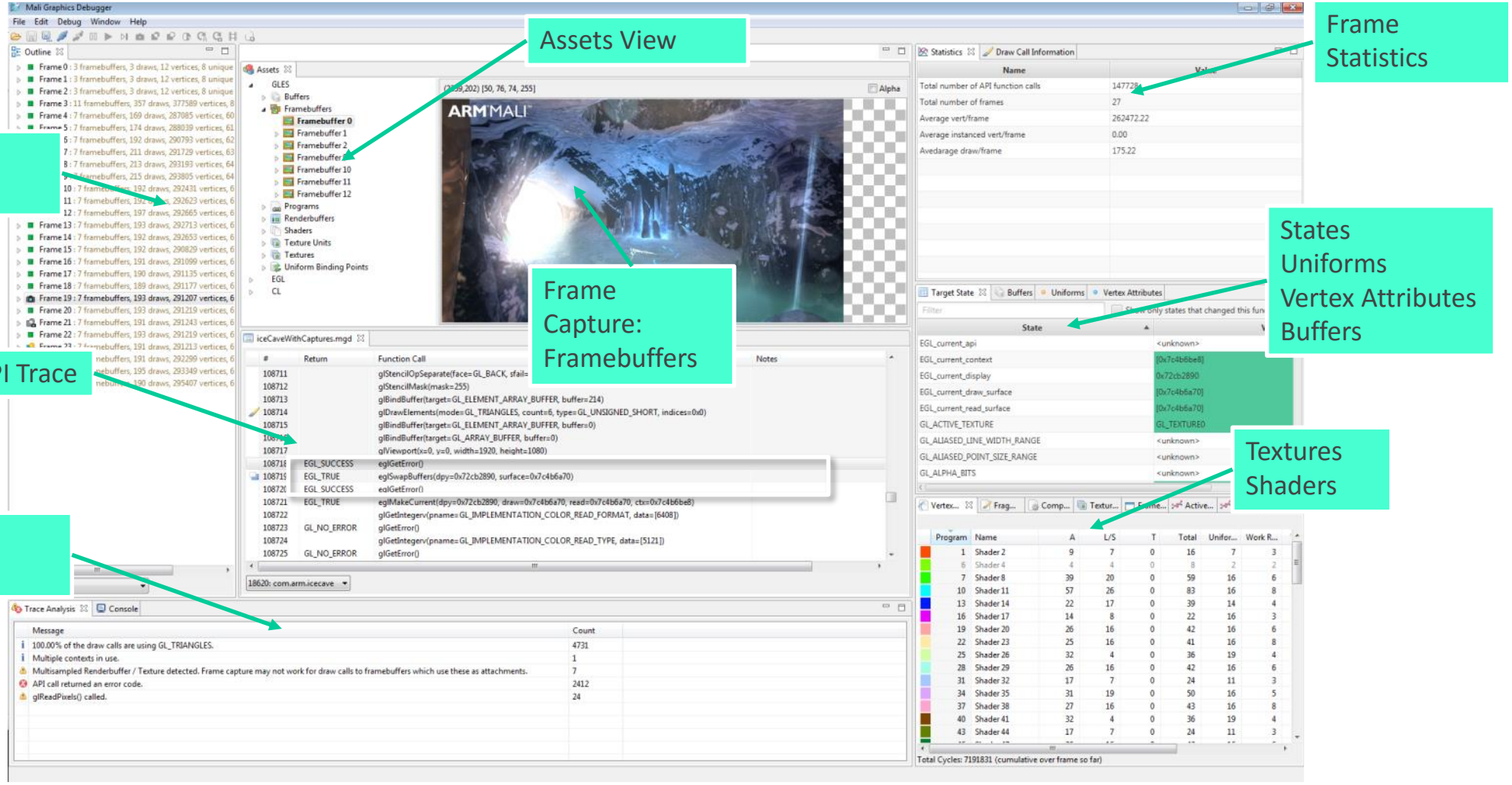
Another tool supporting Vulkan:

Mali Graphics Debugger is an advanced API tracer tool for Vulkan, OpenGL ES, EGL and OpenCL. It allows developers to trace their graphics and compute applications to debug issues and analyze the performance.



- **Vulkan Support**
  - Trace all the function calls in the SPEC.
  - Allows you to see exactly what calls compose your application.
  - Contact the Mali forums and we would love to get you setup. https://community.arm.com/groups/arm-mali-graphics

Investigation with the Mali Graphics Debugger



References

- **SPIRV-Cross**
  - https://github.com/KhronosGroup/SPIRV-Cross
- **Glslang**
  - https://github.com/KhronosGroup/glslang
- **Shaderc**
  - https://github.com/google/shaderc
- **SPIRV-Tools**
  - https://github.com/KhronosGroup/SPIRV-Tools
- **Mali Graphics Debugger**
  - http://malideveloper.arm.com/resources/tools/mali-graphics-debugger/

Lunch!

Have a look at demos, show us your code and return at 1:15pm for part II

Feeding Your Shaders

ARM

Jesse Barker
Principal Software Engineer

Moving to Vulkan: How to make your 3D graphics more explicit

May 26, 2016

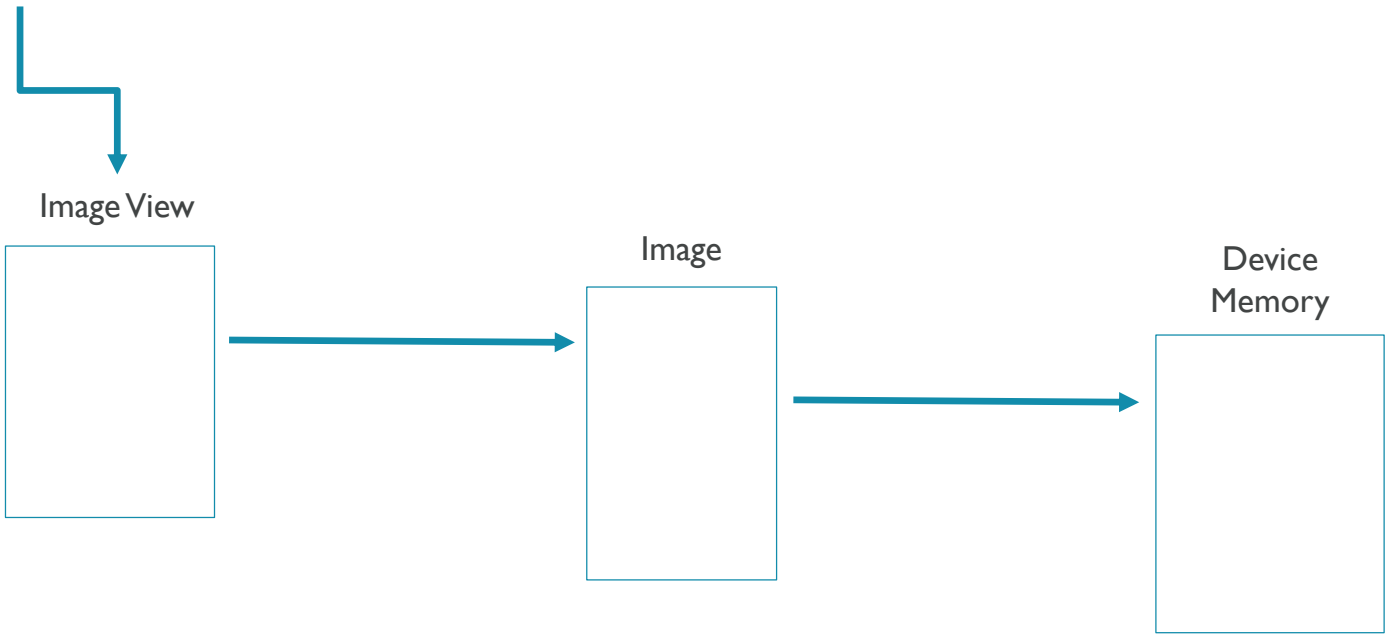
What is a Vulkan Resource?

- Shader Input/Output
- Referenced via *Descriptors*
- Some are specialized in the hardware
  - Vertex Input Attributes
  - Render Targets

- Buffers
- Images
- Samplers
- Input Attachments

What are Vulkan Descriptors?

Handle	Type
myImageView	SAMPLED_IMAGE



What are Descriptor Sets?

```
// uniform blocks:
layout(set = 0, binding = 0) uniform Type0 { ... } ubo0;

// textures:
layout(set = 0, binding = 1) uniform sampler2D tex0;

// SSBO:
layout(set = 0, binding = 2) buffer Type2 { ... } ssbo0;

void main()
    // ...
}
```

binding	type	stages
0	Uniform Buffer	Graphics
1	Image/Sampler	Graphics
2	Storage Buffer	Graphics

What is a Descriptor Pool?

- Parent object of a Descriptor Set
- Allows Descriptor Set management to be threaded
- Manages memory for hardware descriptors

```
typedef struct VkDescriptorPoolSize {
    VkDescriptorType    type;
    uint32_t            descriptorCount;
} VkDescriptorPoolSize;

typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorPoolCreateFlags     flags;
    uint32_t                        maxSets;
    uint32_t                        poolSizeCount;
    const VkDescriptorPoolSize*     pPoolSizes;
} VkDescriptorPoolCreateInfo;
```

Allocating Descriptor Sets

- Define desired layouts of descriptors
- Ask the Descriptor Pool to allocate a Descriptor Set per layout

What is a Pipeline Layout?

```
// uniform blocks:
layout(set = 0, binding = 0) uniform Type0
{ ... } ubo0;
layout(set = 0, binding = 0) uniform Type1
{ ... } ubo1;

// textures:
layout(set = 0, binding = 1) uniform
sampler2D tex0;
layout(set = 1, binding = 0) uniform
sampler2D tex1;

// SSBO:
layout(set = 1, binding = 1) buffer Type2 {
... } ssbo0;

void main() {
    // ...
}
```

Descriptor Set 0

binding	type	stages
0	Uniform Buffer	Graphics
0	Uniform Buffer	Graphics
1	Image/Sampler	Graphics

Descriptor Set 1

binding	type	stages
0	Image/Sampler	Graphics
1	Storage Buffer	Graphics

How do Descriptors get into Descriptor Sets?

```
VKAPI_ATTR void VKAPI_CALL vkUpdateDescriptorSets(
    VkDevice                    device,
    uint32_t
descriptorWriteCount,
    const VkWriteDescriptorSet* pDescriptorWrites,
    uint32_t                    descriptorCopyCount,
    const VkCopyDescriptorSet* pDescriptorCopies);
```

```
typedef struct VkWriteDescriptorSet {
    VkStructureType                  sType;
    const void*                      pNext;
    VkDescriptorSet                  dstSet;
    uint32_t                         dstBinding;
    uint32_t                         dstArrayElement;
    uint32_t                         descriptorCount;
    VkDescriptorType                 descriptorType;
    const VkDescriptorImageInfo*  pImageInfo;
    const VkDescriptorBufferInfo* pBufferInfo;
    const VkBufferView*              pTexelBufferView;
} VkWriteDescriptorSet;

typedef struct VkCopyDescriptorSet {
    VkStructureType    sType;
    const void*        pNext;
    VkDescriptorSet    srcSet;
    uint32_t           srcBinding;
    uint32_t           srcArrayElement;
    VkDescriptorSet    dstSet;
    uint32_t           dstBinding;
    uint32_t           dstArrayElement;
    uint32_t           descriptorCount;
} VkCopyDescriptorSet;
```

Finally, I'm ready to use my Descriptor Sets

```
VKAPI_ATTR void VKAPI_CALL vkCmdBindDescriptorSets(
    VkCommandBuffer         commandBuffer,
    VkPipelineBindPoint     pipelineBindPoint,
    VkPipelineLayout        layout,
    uint32_t                firstSet,
    uint32_t                descriptorSetCount,
    const VkDescriptorSet*  pDescriptorSets,
    uint32_t                dynamicOffsetCount,
    const uint32_t*         pDynamicOffsets);
```

- Bound sets must match pipeline layout
- Graphics or compute?
- Simple layout is best

What about Vertex Input?

Vertex Input Description

If your shader declares:

```
in vec3 position;
in uvec2 texcoord;
```

Your C code declares:

```
struct Position
{
    float x, y, z;
};

struct Texcoord
{
    uint8_t u, v;
};
```

```
const VkVertexInputBindingDescription binding[] =
{
    {
        0,                                          // binding
        sizeof(float) * 3,                          // stride
        VK_VERTEX_INPUT_RATE_VERTEX                 // inputRate
    },
    {
        1,                                          // binding
        sizeof(uint8_t) * 2,                        // stride
        VK_VERTEX_INPUT_RATE_VERTEX                 // inputRate
    },
};

const VkVertexInputAttributeDescription attributes[] =
{
    {
        0,                                          // location
        binding[0].binding,                         // binding
        VK_FORMAT_R32G32B32_SFLOAT,                 // format
        0                                           // offset
    },
    {
        1,                                          // location
        binding[1].binding,                         // binding
        VK_FORMAT_R8G8_UNORM,                       // format
        0                                           // offset
    }
};
```

Questions?

ARM

SAMSUNG

KHRONOS GROUP

Vulkan Subpasses
or
The Frame Buffer is Lava

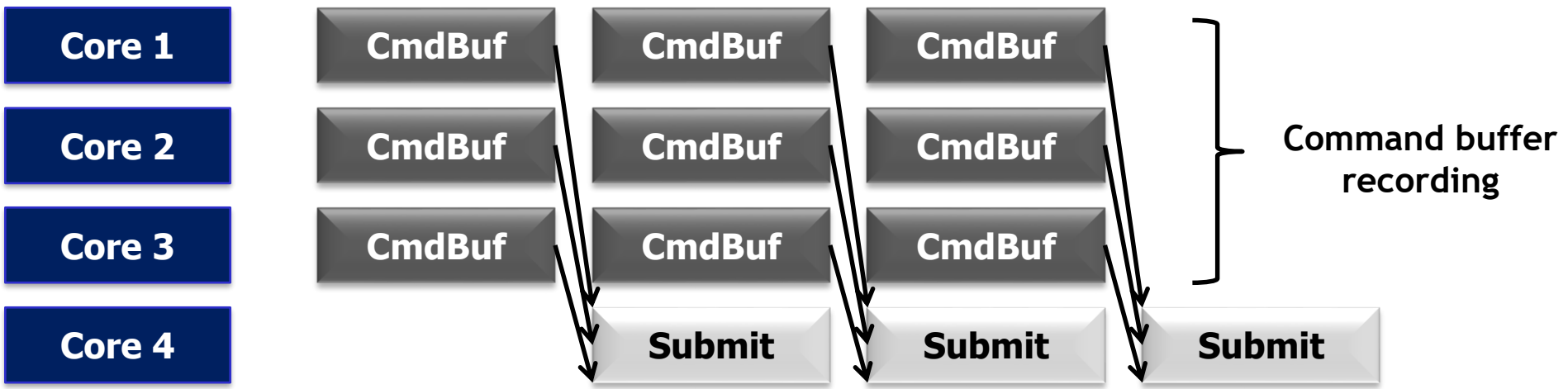
Andrew Garrard
Samsung R&D Institute UK

Vulkan: Making use of the GPU more efficient

- Vulkan aims to reduce the overheads of keeping the GPU busy

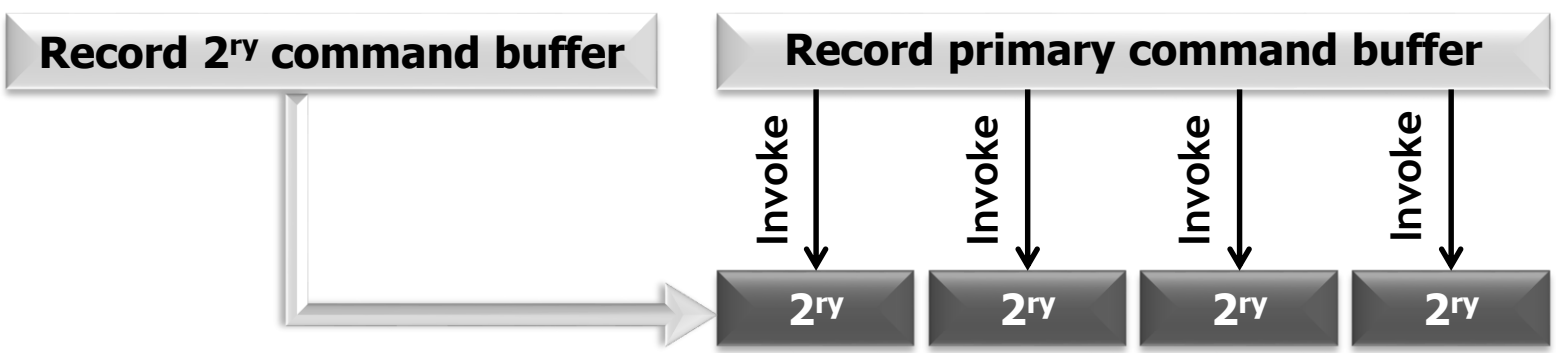
Vulkan: Making use of the GPU more efficient

- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores



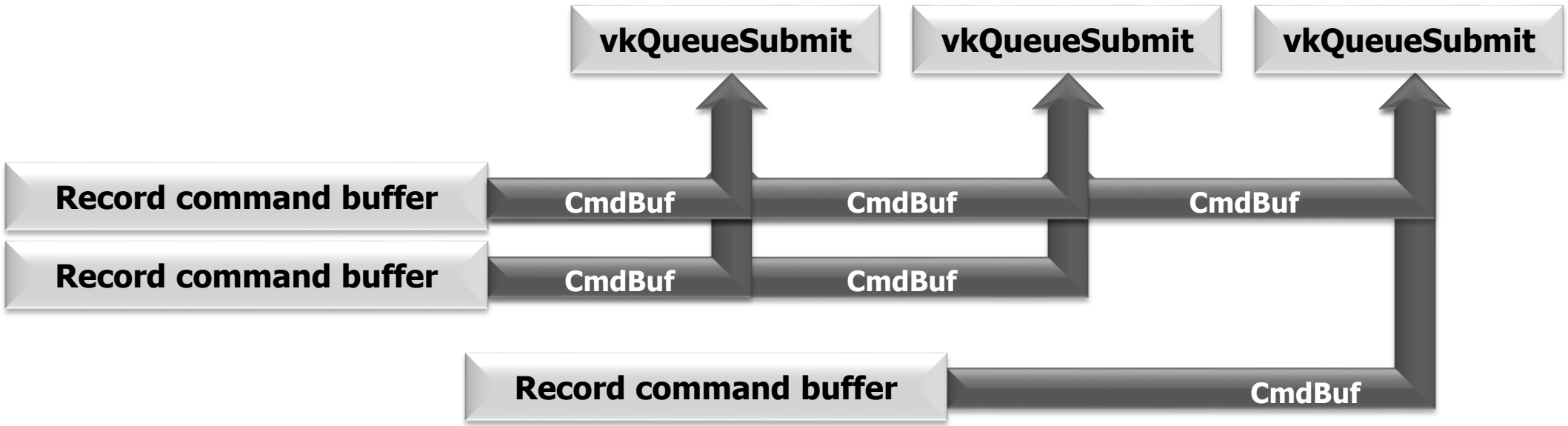
Vulkan: Making use of the GPU more efficient

- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time



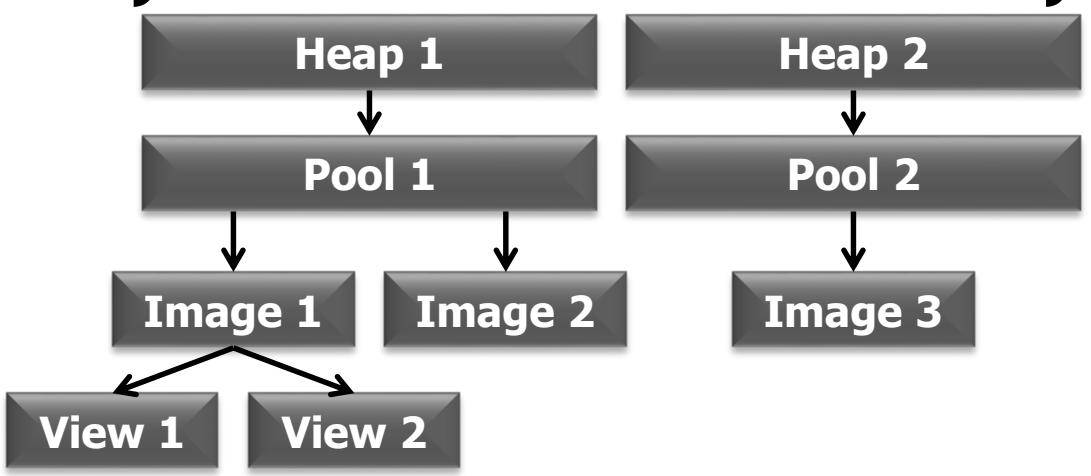
Vulkan: Making use of the GPU more efficient

- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time



Vulkan: Making use of the GPU more efficient

- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
  - Potentially more efficient memory management



User-defined memory reuse

Explicit state transitions

Cost invoked at defined points

Vulkan: Making use of the GPU more efficient

- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
  - Potentially more efficient memory management
  - Avoiding unpredictable shader compilation

Vulkan: Making use of the GPU more efficient

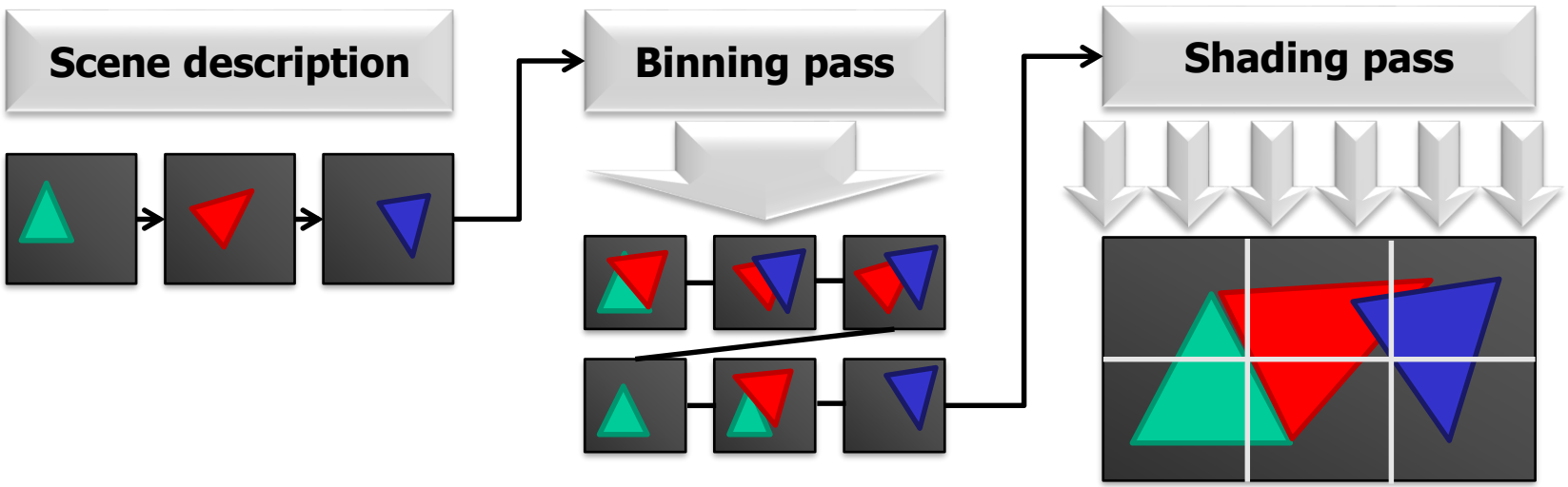
- **Vulkan aims to reduce the overheads of keeping the GPU busy**
  - Efficient generation of work on multiple CPU cores
  - Reuse of command buffers to avoid CPU build time
  - Potentially more efficient memory management
  - Avoiding unpredictable shader compilation
- **Mostly, the message has been that if you're entirely limited by shader performance or bandwidth, Vulkan can't help you (there is no magic wand)**

Vulkan: Making ~~use of~~ the GPU more efficient

- Actually, that's not entirely true...
- APIs like OpenGL were designed when the GPU looked very different (or was partly software)
- The way to design an efficient mobile GPU is not a perfect match for OpenGL
  - Think a CPU's command decode unit/microcode
- But the translation isn't always perfectly efficient

Tiled GPUs

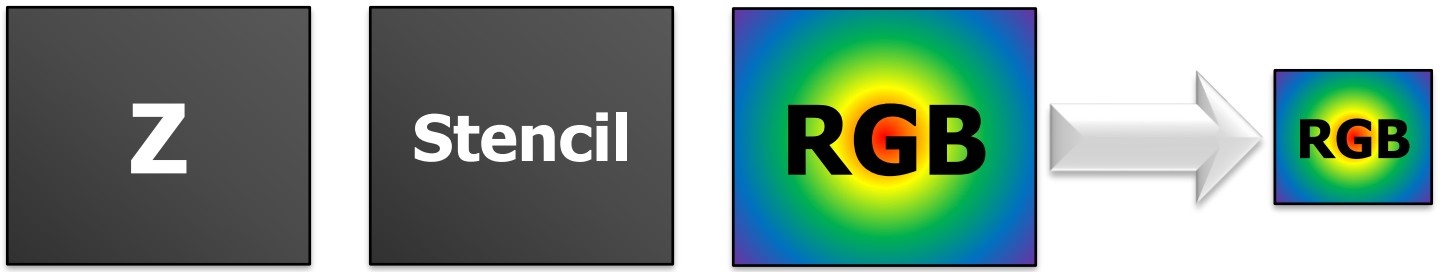
- **Most (not all) mobile GPUs use tiling**
  - It's all about the bandwidth (size and power limits)



- **On-chip tile memory is much faster than the main frame buffer**

Not everything reaches memory

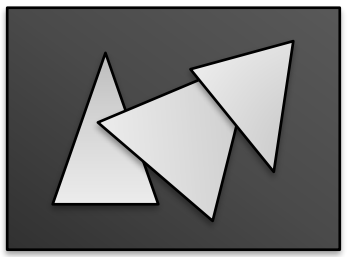
- **Rendering requires lots of per-pixel data**
  - Z, stencil
  - Full multisample resolution
- **We usually only care about the final image**

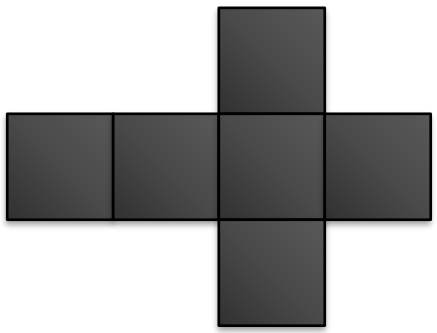


  - We can throw away Z and stencil
  - We only need a downsampled (A)RGB
  - Don’t need to load anything from a previous frame

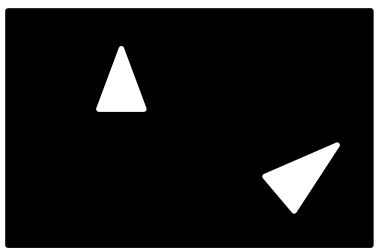
Sometimes we want the results of rendering

- Output from one rendering job can be used by the next
- Z buffer for shadow maps





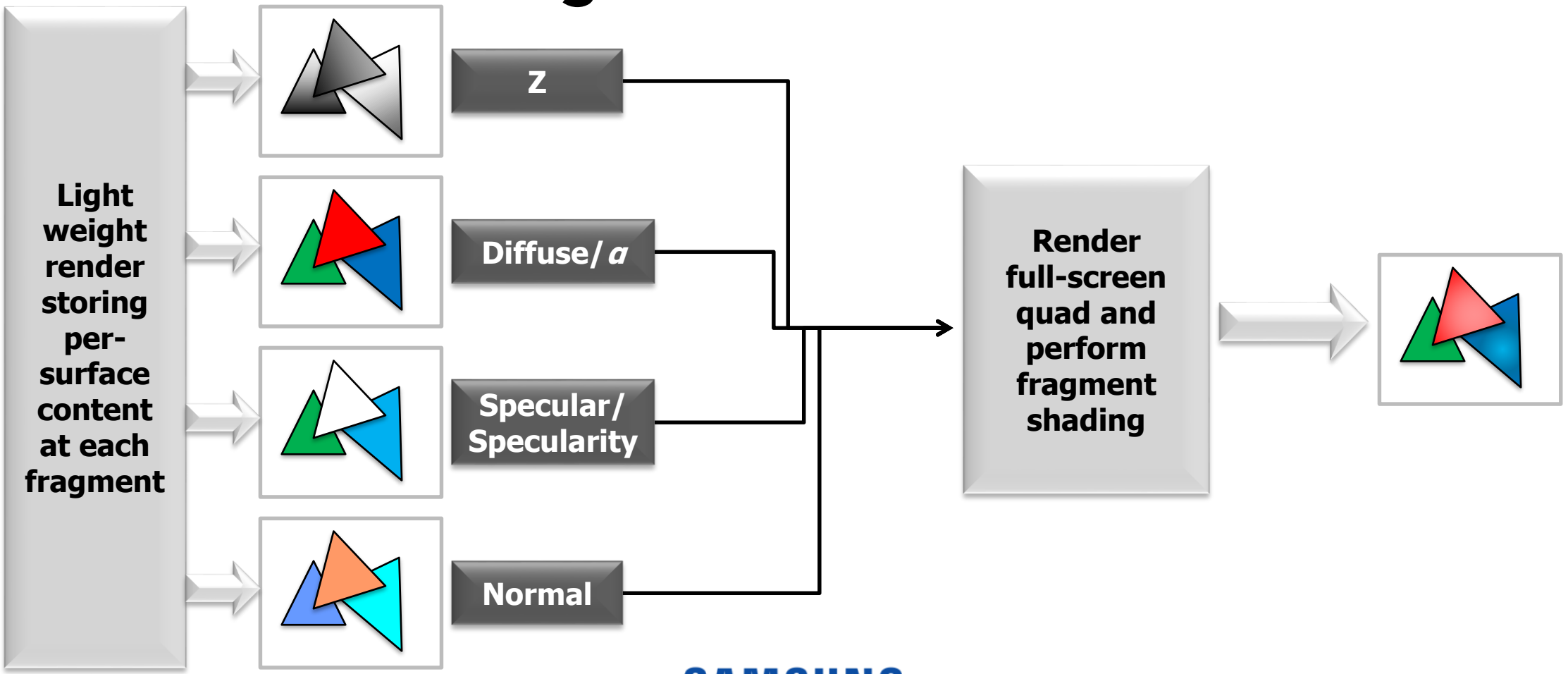
- Rendering for environment maps
- HDR bloom



- These can have low resolution and may not take much bandwidth

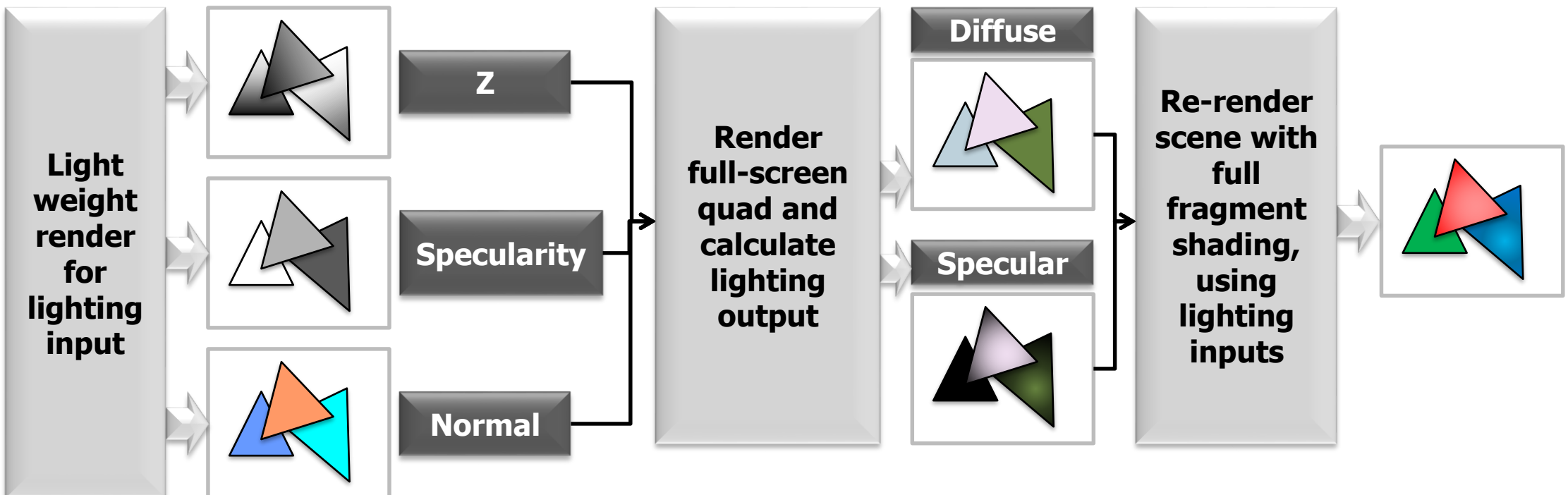
Sometimes you *do* need framebuffer resolution

- Deferred shading



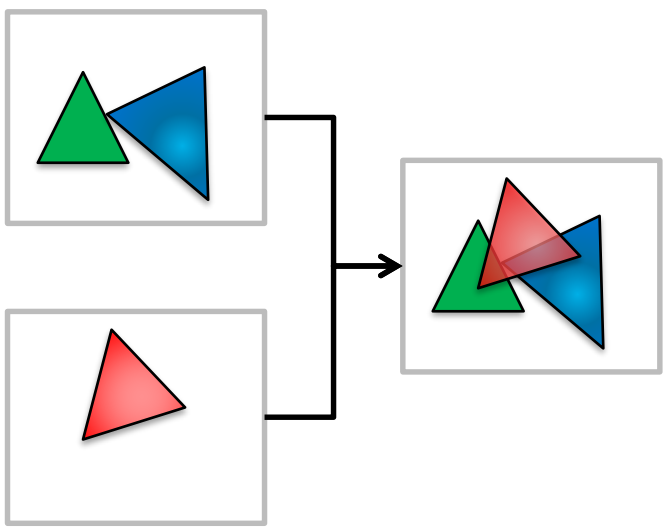
Sometimes you *do* need framebuffer resolution

- Deferred shading
- Deferred lighting



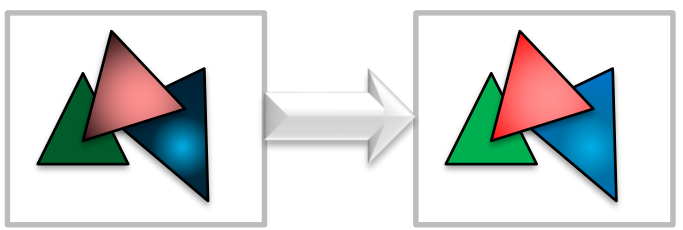
Sometimes you *do* need framebuffer resolution

- Deferred shading
- Deferred lighting
- Order-independent transparency



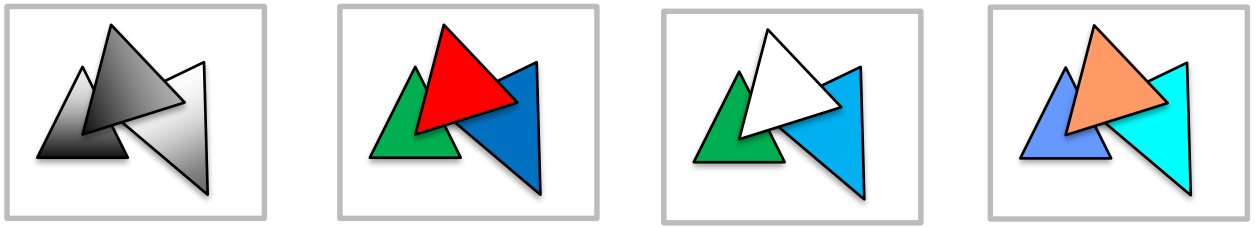
Sometimes you *do* need framebuffer resolution

- Deferred shading
- Deferred lighting
- Order-independent transparency
- HDR tone mapping



Rendering outputs separately

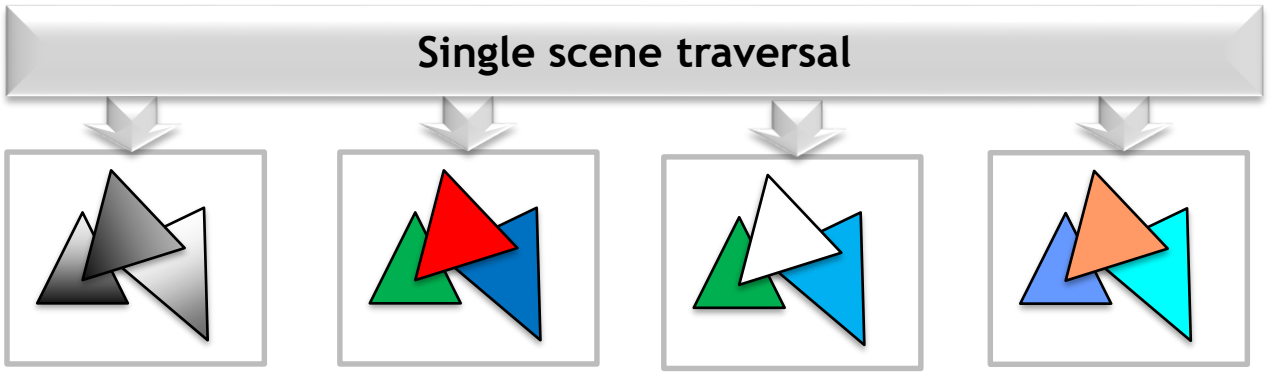
- **Rendering to each surface separately is *bad***



- **Geometry has a per-bin cost**
  - Sometimes the cost is low, but it's there
  - Vertices in multiple bins get processed repeatedly
  - Rendering the scene repeatedly is painful
- **Even immediate-mode renderers hate this!**

Multiple render targets don't help much

- **Using MRTs means multiple buffers in one pass**

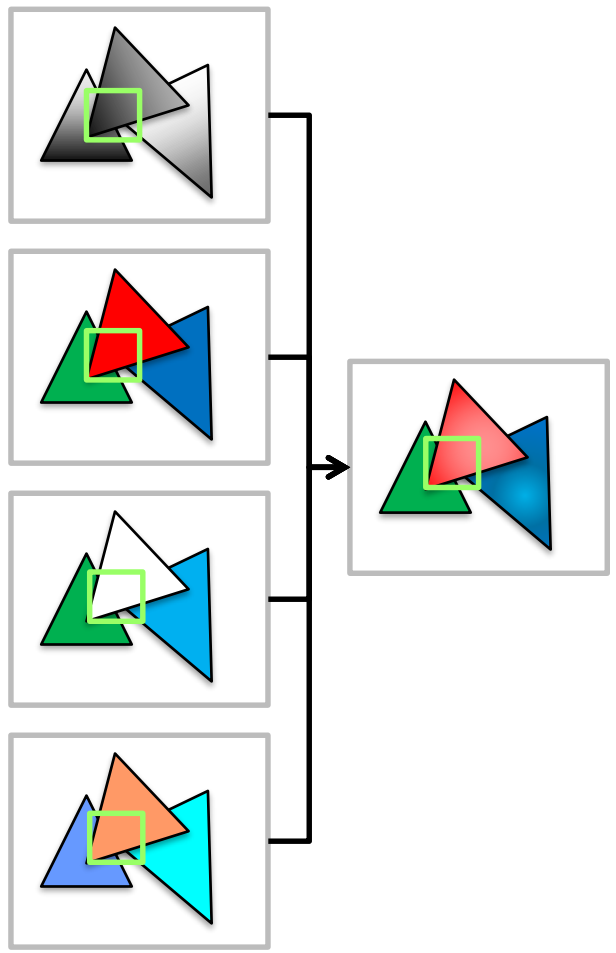


This is a typical approach for immediate-mode renderers (e.g. desktop/console systems)

- **Reduces the geometry load (only process once)**
- **Still writing a *lot* of data off-chip**
  - Tilers are all about trying not to do this!
  - Increases use of shader resources may slow some h/w

Pixel Local Storage (OpenGL ES extension)

- **Tiler-friendly (at last)**
  - Store only the current tile values
  - Read them later in the tile processing
- **But not portable!**
  - Not practical on immediate renderers
  - Debugging on desktop won't work!
  - Capabilities vary between devices
  - Driver doesn't have visibility
  - Data access is restricted

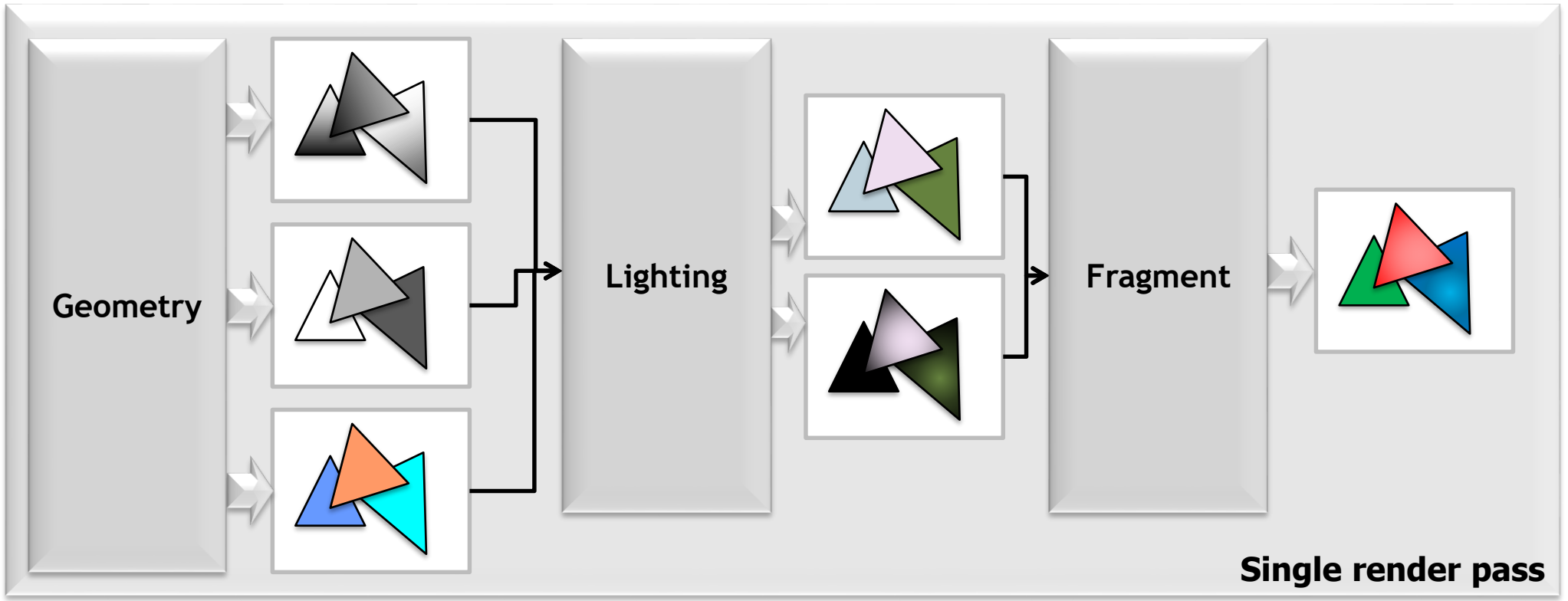


Vulkan: Explicit dependencies

- **Vulkan has direct support for this type of rendering work load**
- **By telling the driver how you intend to use the rendered results, the driver can produce a better mapping to the hardware**
  - The extra information is a little verbose, but simpler than handling all possible cases yourself!

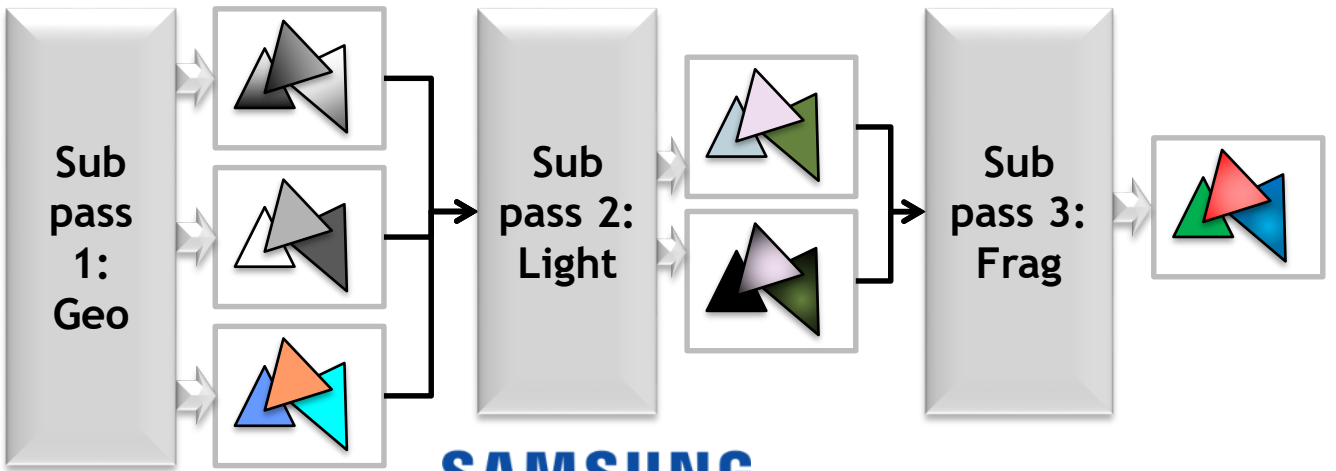
Vulkan render passes and subpasses

- **A render pass groups dependent operations**
  - All images written in a render pass are the same size



Vulkan render passes and subpasses

- **A render pass groups dependent operations**
  - All images written in a render pass are the same size
- **A render pass contains a number of *subpasses***
  - Subpasses describe access to *attachments*
  - Dependencies can be defined between subpasses



Vulkan render passes and subpasses

- **A render pass groups dependent operations**
  - All images written in a render pass are the same size
- **A render pass contains a number of *subpasses***
  - Subpasses describe access to *attachments*
  - Dependencies can be defined between subpasses
- **Each render pass instance has to be contained within a single command buffer (unit of work)**
  - Some tilers schedule by render pass

Defining a render pass

- **VkRenderPassCreateInfo**
  - VkAttachmentDescription *pAttachments
    - Just the descriptions, not the actual attachments!
  - VkSubpassDescription *pSubpasses
  - VkSubpassDependency *pDependencies
- **vkCreateRenderPass(device, createInfo,.. pass)**
  - Gives you a VkRenderPass object
  - This is a *template* that you can use repeatedly
    - When we use it, we get a *render pass instance*

Describing attachments for a render pass

- **VkAttachmentDescription**
  - format/samples
  - loadOp
    - VK_ATTACHMENT_LOAD_OP_LOAD to preserve
    - VK_ATTACHMENT_LOAD_OP_DONT_CARE for overwrites
    - VK_ATTACHMENT_LOAD_OP_CLEAR uniform clears (e.g. Z)
  - storeOp
    - VK_ATTACHMENT_STORE_OP_STORE to output it
    - VK_ATTACHMENT_STORE_OP_DONT_CARE may discard after the render pass

Defining a subpass

- **VkSubpassDescription**
  - pInputAttachments
    - Which of the render pass's attachments this subpass reads
  - pColorAttachments
    - Which ones this subpass writes (1:1 - optional)
  - pResolveAttachments
    - Which ones this subpass writes (resolving multisampling)
  - pPreserveAttachments
    - Which attachments need to persist across this subpass
  - Subpasses are numbered and ordered

Defining subpass dependencies

- **VkSubpassDependency**
  - srcSubpass
  - dstSubpass
    - Where the dependency applies (can be external)
  - srcStageMask
  - dstStageMask
    - Execution dependencies between subpasses
  - srcAccessMask
  - dstAccessMask
    - Memory dependencies between subpasses

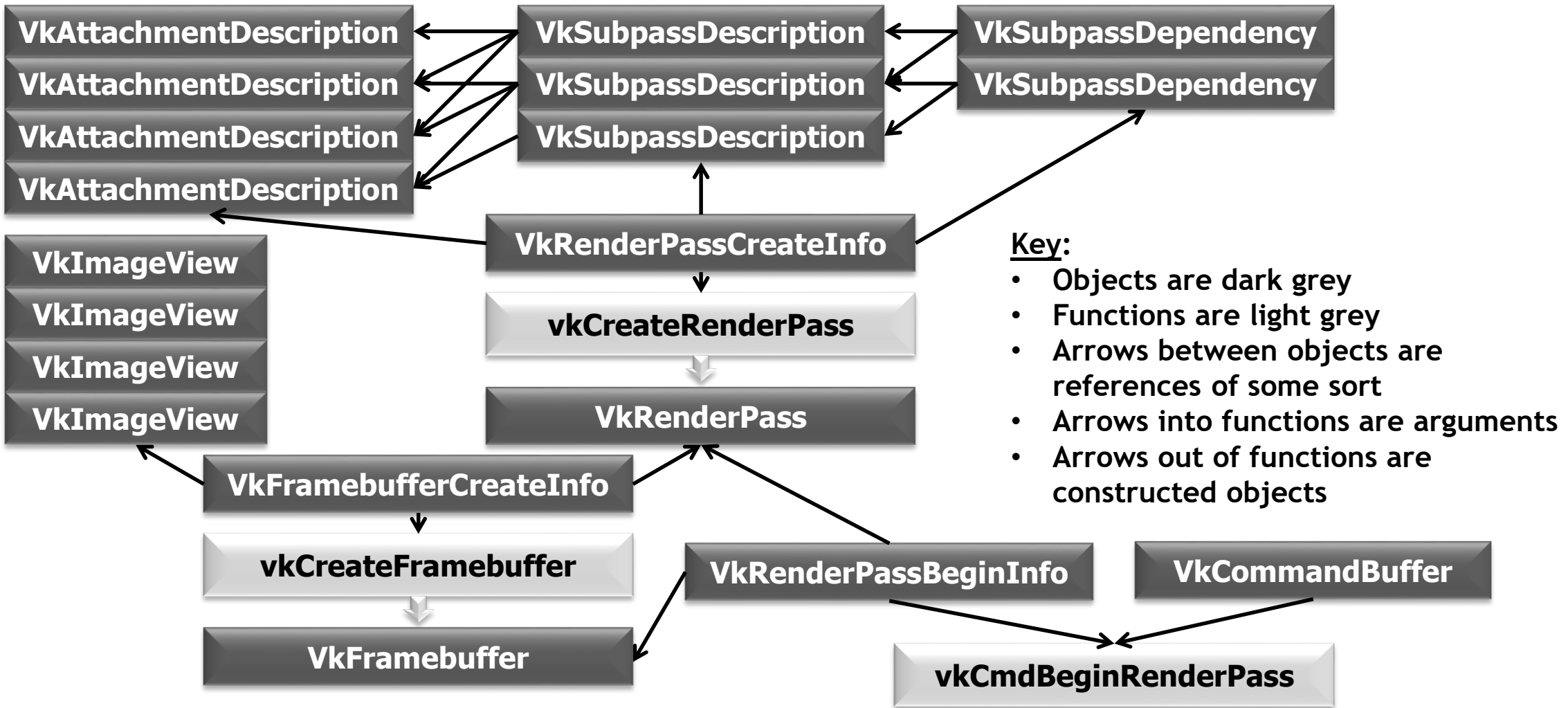
Vulkan framebuffers

- **A VkFramebuffer defines the set of attachments used by a render pass instance**
- **VkFramebufferCreateInfo**
  - renderPass
  - pAttachments
    - These are actual VkImageViews this time!
  - width
  - height
  - layers

Starting to use a render pass

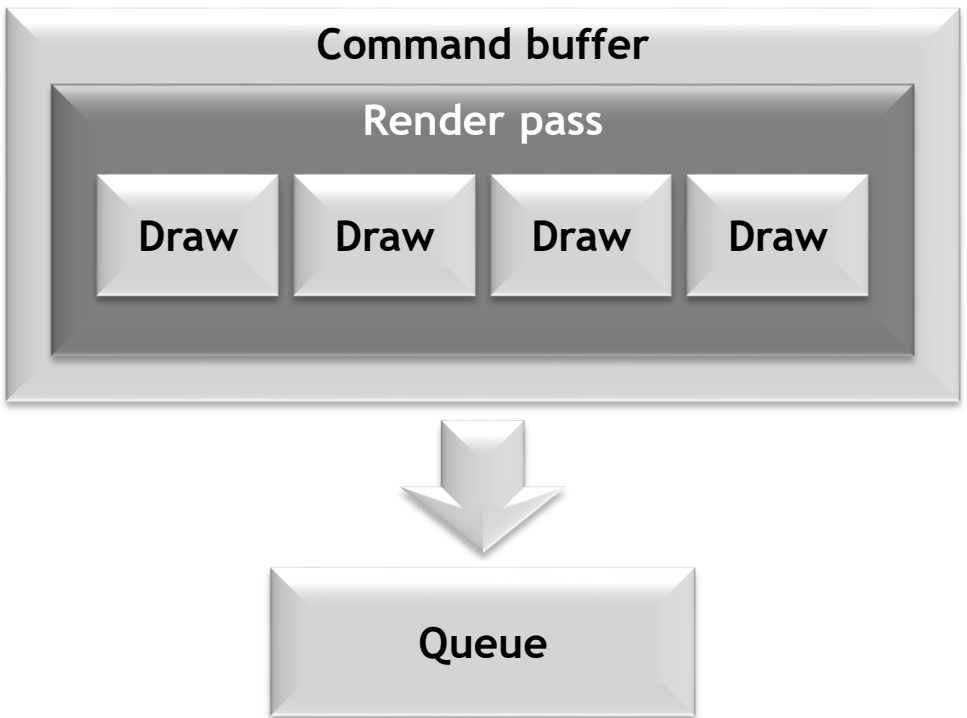
- **vkCmdBeginRenderPass/vkCmdEndRenderPass**
  - Starts a render pass *instance* in a command buffer
    - You start in the first (maybe only) subpass implicitly
  - pRenderPassBegin contains configuration
- **VkRenderPassBeginInfo**
  - VkRenderPass renderPass
    - The render pass “template”
  - VkFrameBuffer framebuffer
    - Specifies targets for rendering

Putting it all together…



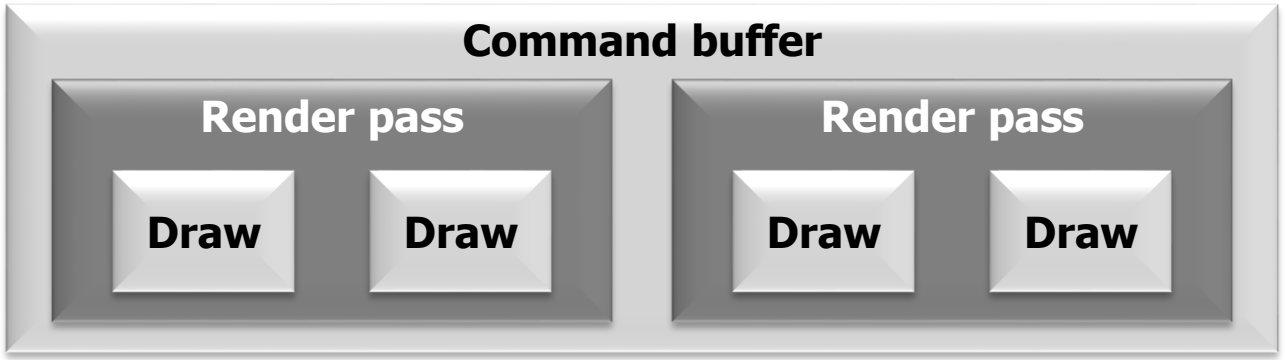
Simple rendering

- vkAllocateCommandBuffers (VK_COMMAND_BUFFER_LEVEL_PRIMARY)
- vkBeginCommandBuffer
- vkCmdBeginRenderPass
- vkCmdDraw (etc.)
- vkCmdEndRenderPass
- vkEndCommandBuffer
- vkQueueSubmit



Multiple render passes

- **You can have more than one render pass in a command buffer**
  - Yes, Leeloo multipass, we know…



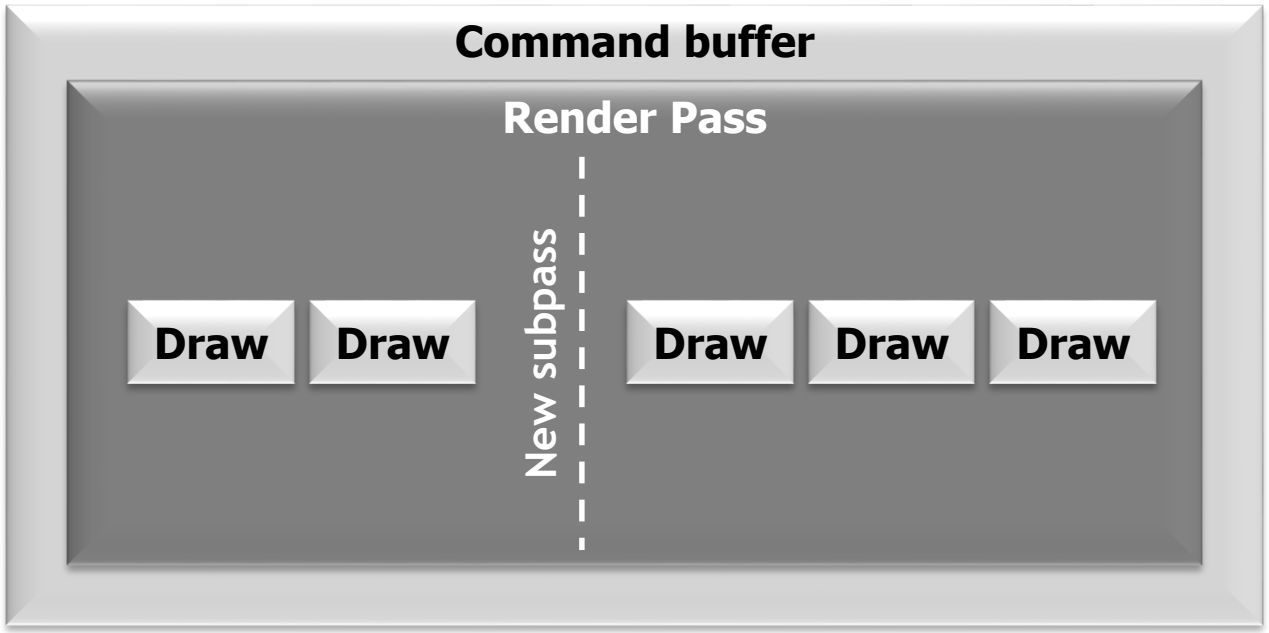
  - So a command buffer can render to many outputs
    - E.g. you could render to the same shadow and environment maps every frame by reusing the same command buffer
  - But it must be the *same* outputs each time you submit
    - A specific render pass instance has fixed vkFrameBuffers!

Two limitations…

- **Different render passes ⇒ independent outputs**
  - Rendering goes off-chip, there's no PLS-style on-chip reuse of pixel contents
- **You can't reuse the same command buffer with a different render target**
  - E.g. for double buffering or streamed content
  - We'll come back to this…
- **Still sometimes all you need, though!**

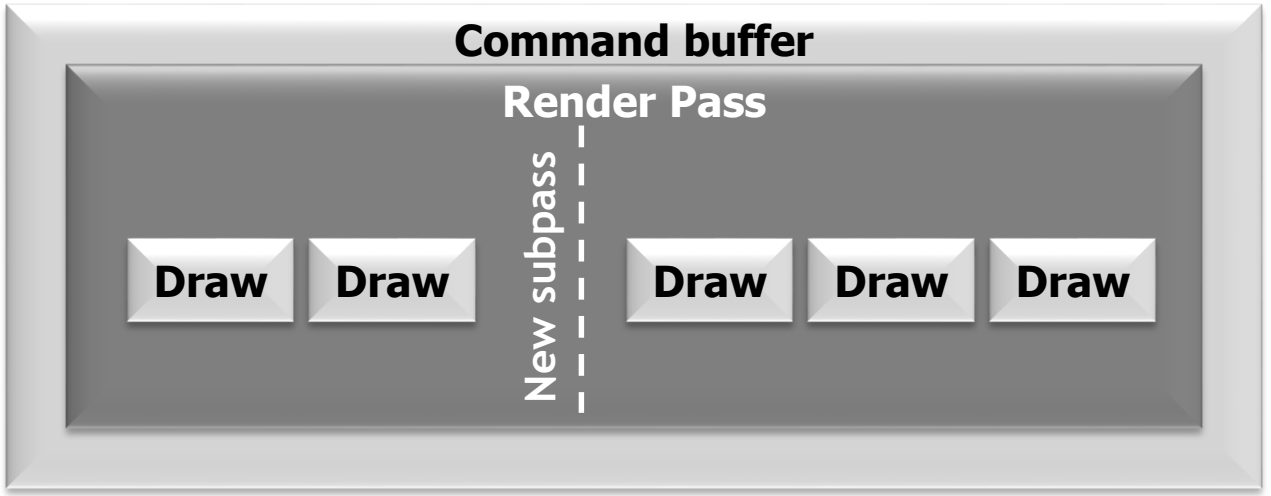
More than one subpass

- **vkCmdNextSubpass moves to the next subpass**
  - Implicitly start in the first subpass of the render pass
  - Dependencies say what you're accessing from previous subpasses
  - Same render pass so accesses stay on chip (if possible)



Using multiple subpasses

- vkCmdBeginCommandBuffer
- vkCmdBeginRenderPass
- vkCmdDraw (etc.)
- vkCmdNextSubpass
- vkCmdDraw (etc.)
- vkCmdEndRenderPass
- vkCmdEndCommandBuffer



Accessing subpass output in fragment shaders

- **In SPIR-V, previous subpass content is read with OpImageRead**
  - Coordinates are sample-relative, and need to be 0
  - OpTypeImage Dim = SubpassData
- **In GLSL (using GL_KHR_vulkan_glsl):**
  - Types for subpass access are [ui]subpassInput(MS)
    - layout(input_attachment_index = i, …) uniform subpassInput t; to select a subpass
  - subpassLoad() to access the pixel

C.f. __pixel_localEXT layouts in EXT_shader_pixel_local_storage when using OpenGL ES

Avoiding unnecessary allocations

- **If we're using subpasses, we likely don't need the images in memory**
  - A tiler may be able to process the subpasses entirely on-chip, without needing an allocation
  - Still need to "do the allocation" in case the tiler can't handle the request/on an immediate-mode renderer!
    - Won't commit resources unless it actually needs to
- **vkCreateImage flags for "lazy committal"**
  - VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT

Vulkan subpasses: advantages

- **The driver knows what you're doing**
  - It can reorder subpasses
  - It can change the tile size
  - It can balance resources between subpasses
  - *It will fall back to memory for you* if it has to
  - Under the hood, mechanism likely matches PLS
- **Works on immediate mode renderers**
  - Probably MRTs and normal external writes
  - Desktop debugging tools will work!

EXT_shader_pixel_local_storage is actually *more* explicit than Vulkan here (and may still be offered as an extension)

There's more: Secondary command buffers

- **Vulkan has two levels of command buffers**
  - Determined by vkAllocateCommandBuffers
- **VK_COMMAND_BUFFER_LEVEL_PRIMARY**
  - Main command buffer, as we've seen so far
- **VK_COMMAND_BUFFER_LEVEL_SECONDARY**
  - Command buffer that can be invoked from the primary command buffer

Use of secondary command buffers

- **vkBeginCommandBuffer**
  - Takes a VkCommandBufferBeginInfo
- **VkCommandBufferBeginInfo**
  - flags include:
    - VK_COMMANDBUFFER_USAGE_RENDER_PASS_CONTINUE_BIT
  - pInheritanceInfo
- **VkCommandBufferInheritanceInfo**
  - renderPass and subpass
  - framebuffer (can be null, more efficient if known)

Secondary command buffers and passes

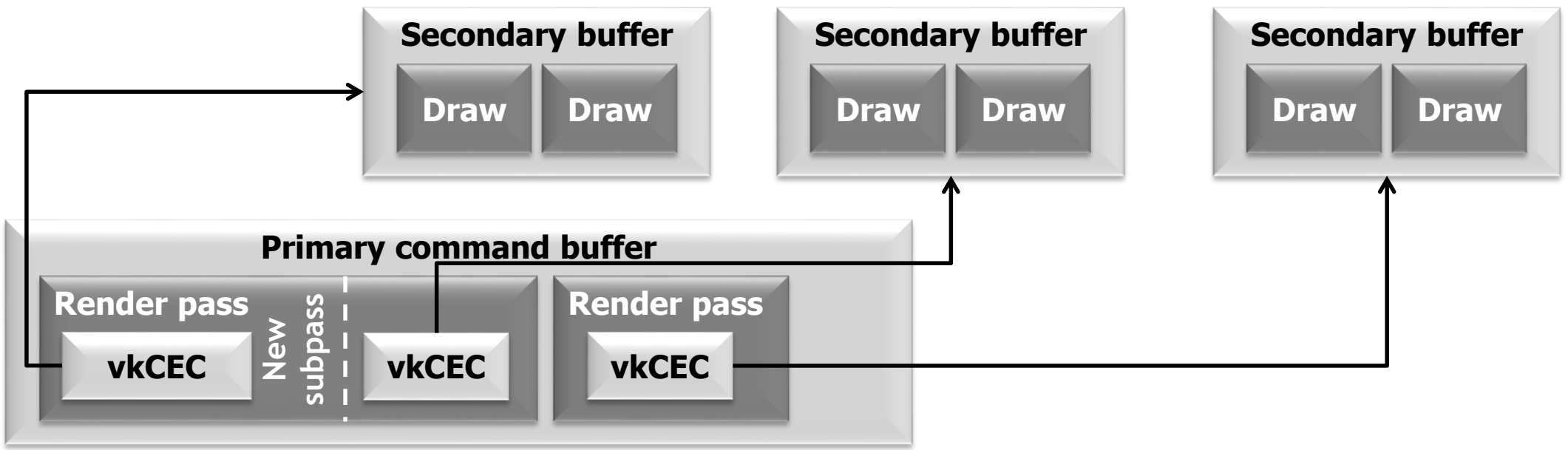
- **Why do we need the “continue bit”?**
  - *Render passes (and subpasses) can’t start in a secondary command buffer*
  - Non-render pass stuff can be in a secondary buffer
    - You *can* run a compute shader outside a render pass
  - Otherwise, the render pass is inherited from the primary command buffer

Secondary command buffers and passes

- **Why specify render pass/framebuffer?**
  - Command buffers needs to know this when recording
    - Some operations depends on render pass info (e.g. format)
  - Framebuffer is optional (can *just* inherit)
    - If you *can* specify the actual framebuffer, the command buffer can be less generic and therefore may be faster

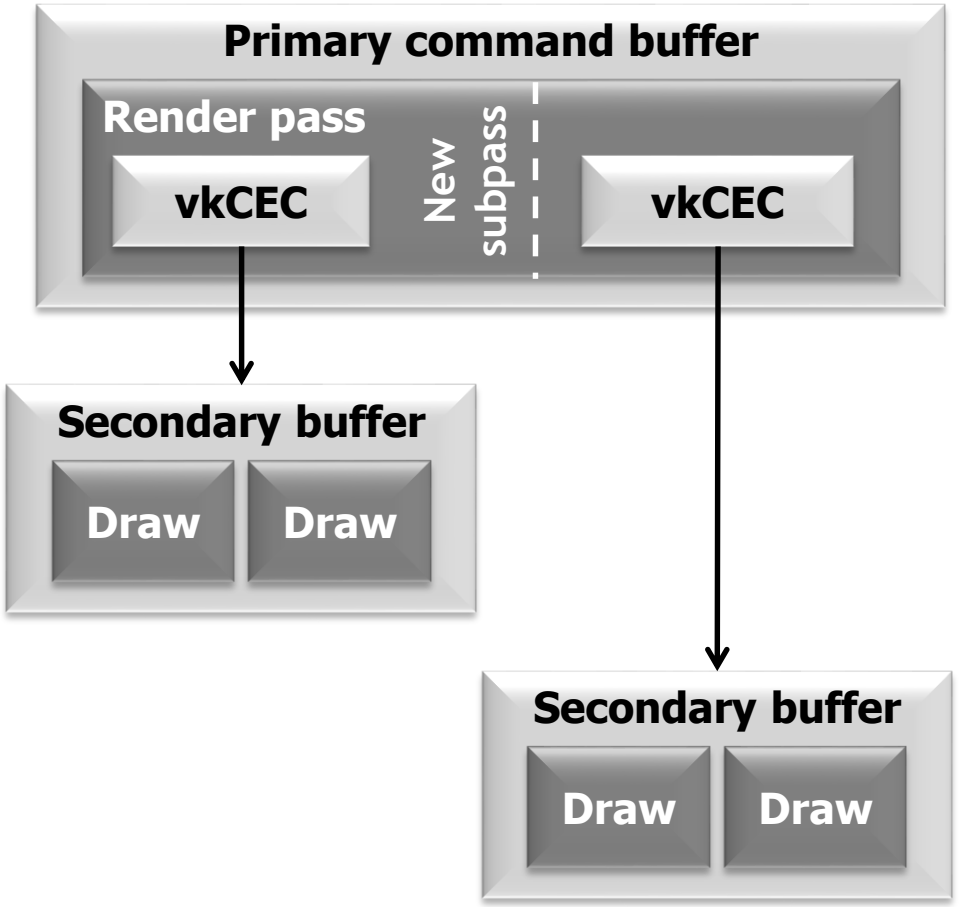
Invoking the secondary command buffer

- You can't submit a secondary command buffer
- You have to invoke it from a primary command buffer with vkCmdExecuteCommands



Secondary command buffer code

- vkCmdBeginCommandBuffer
- vkCmdBeginRenderPass
- vkCmdExecuteCommands
- vkCmdNextSubpass
- vkCmdExecuteCommands
- vkCmdEndRenderPass
- vkCmdEndCommandBuffer



Performance and parallelism

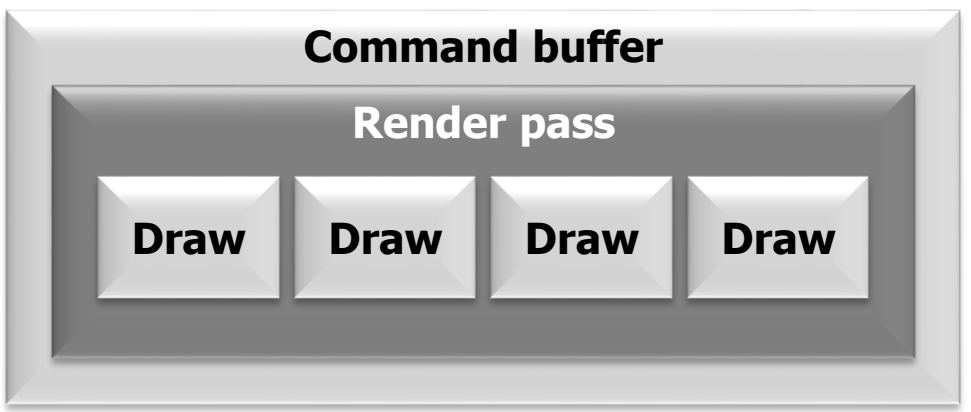
- **Creating a command buffer can be slow**
  - Lots of state to check, may require compilation
    - This happens in GLES as well, you just don't control when!
- **So create secondary command buffers on different threads**
  - Lots of 4- and 8-core CPUs in cell phones these days
- **Invoking the secondary buffer is lightweight**
  - Primary command buffer generation is quick(er)

What does this have to do with passes?

- **Remember:**
  - Render passes exist within (primary) command buffers
    - The command buffer sets up the GPU for the render pass
  - On-chip rendering happens within a render pass
    - If you want content to persist between render passes, it'll reach memory (or at least cache), not stay in the tile buffer
  - You can't use multiple threads to build work for a primary command buffer in parallel
    - You *can* build many secondary command buffers at once

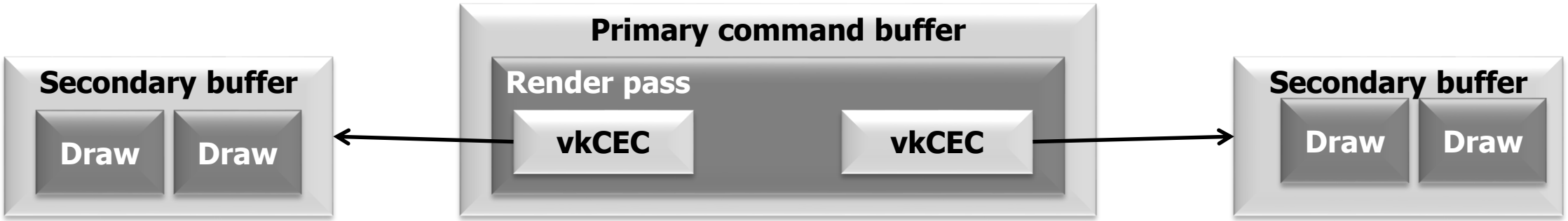
You can’t mix and match

- **Within a subpass you can either (but not both):**
  - Execute rendering commands directly in the primary command buffer
    - VK_SUBPASS_CONTENTS_INLINE



You can’t mix and match

- **Within a subpass you can either (but not both):**
  - Execute rendering commands directly in the primary command buffer
    - VK_SUBPASS_CONTENTS_INLINE
  - Invoke secondary command buffers from the primary command buffer with vkCmdExecuteCommands
    - VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS



You can’t mix and match

- **Within a subpass you can either (but not both):**
  - Execute rendering commands directly in the primary command buffer
    - VK_SUBPASS_CONTENTS_INLINE
  - Invoke secondary command buffers from the primary command buffer with vkCmdExecuteCommands
    - VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS
  - Chosen by vkCmdBeginRenderPass/vkCmdNextSubpass
    - Remember: you can only do these in a primary command buffer!

Command buffer reuse: even faster

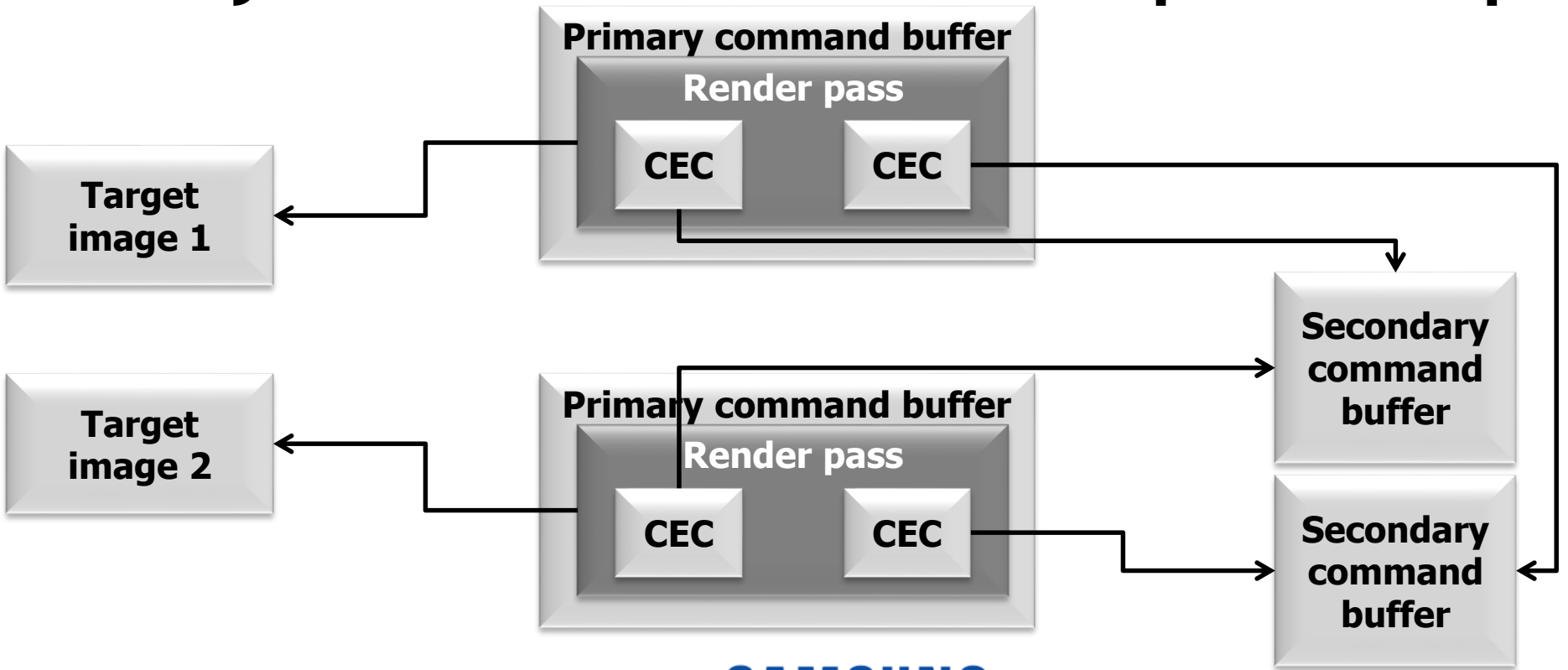
- **Primary command buffers work with a fixed render pass and framebuffer**
  - You can reuse a primary command buffer, but it will always access the same images – often good enough
    - May have to wait for execution to end; can't be "one-time"
- **What if you want to access different targets?**
  - E.g. a cycle of framebuffers or streamed content?
  - You *can* round-robin several command buffers
  - Or you can use secondary command buffers!

Compatible render passes and frame buffers

- **The render pass a secondary command buffer uses needn't be the one it was recorded with**
  - It can be "compatible"
    - Same formats, number of sub-passes, etc.
- **You can have primary command buffers with different outputs, and they can re-use secondary command buffers**
  - The primary has to be different to record new targets
  - The primary may have to patch secondary addresses

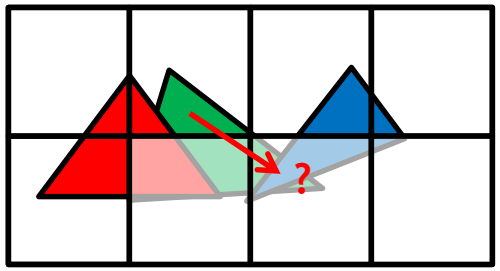
Almost-free use with changing framebuffers

- No cost for secondary command buffers
- Primary command buffer is simple and quick



So I can do bloom/DoF/rain/motion blur…!

- **No! Remember, you can only access the current pixel**



- **Tilers process one tile at a time**
  - If you could try to access a different pixel, the tile containing it may not be there
  - You have to write out the whole image to do this
    - Slow, painful, last resort!
  - Yes, we can think of possible solutions too
    - Give it time (lots of different hardware out there)

Coming out of the shadow(buffer)s

- **Render passes are integral to the Vulkan API**
  - Reflects modern, high-quality rendering approaches
- **The driver has more information to work with**
  - It can do more for you
    - Remember this if you complain it's verbose!
- **Hardware resource management is *hard***
  - Expect drivers to get better over time
- **Another tool for better mobile gaming**

Thank you

- Over to you…

Andrew Garrard

a.garrard at samsung.com

Keeping your GPU fed without getting bitten

Tobias Hector
May 2016

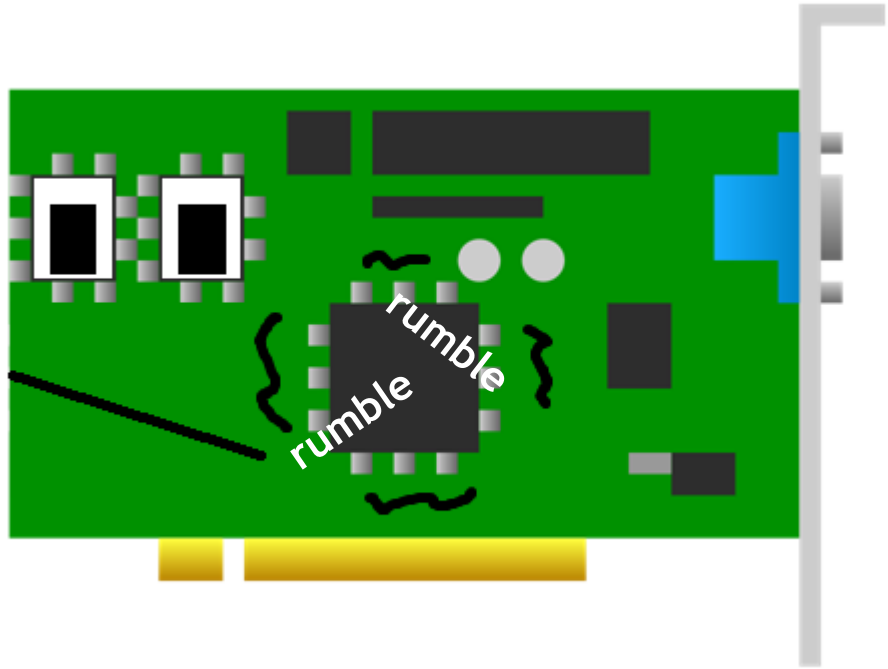
Introduction

- **You have delicious draw calls**
  - Yummy!

Introduction

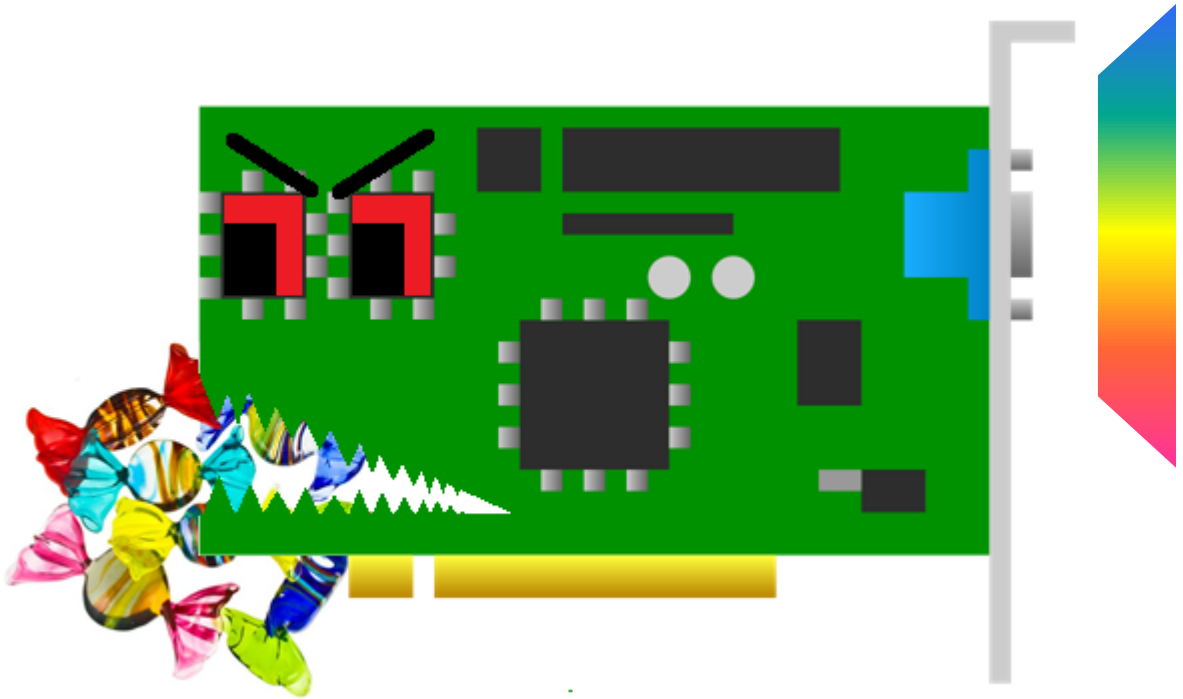
- **You have delicious draw calls**
  - Yummy!

- **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



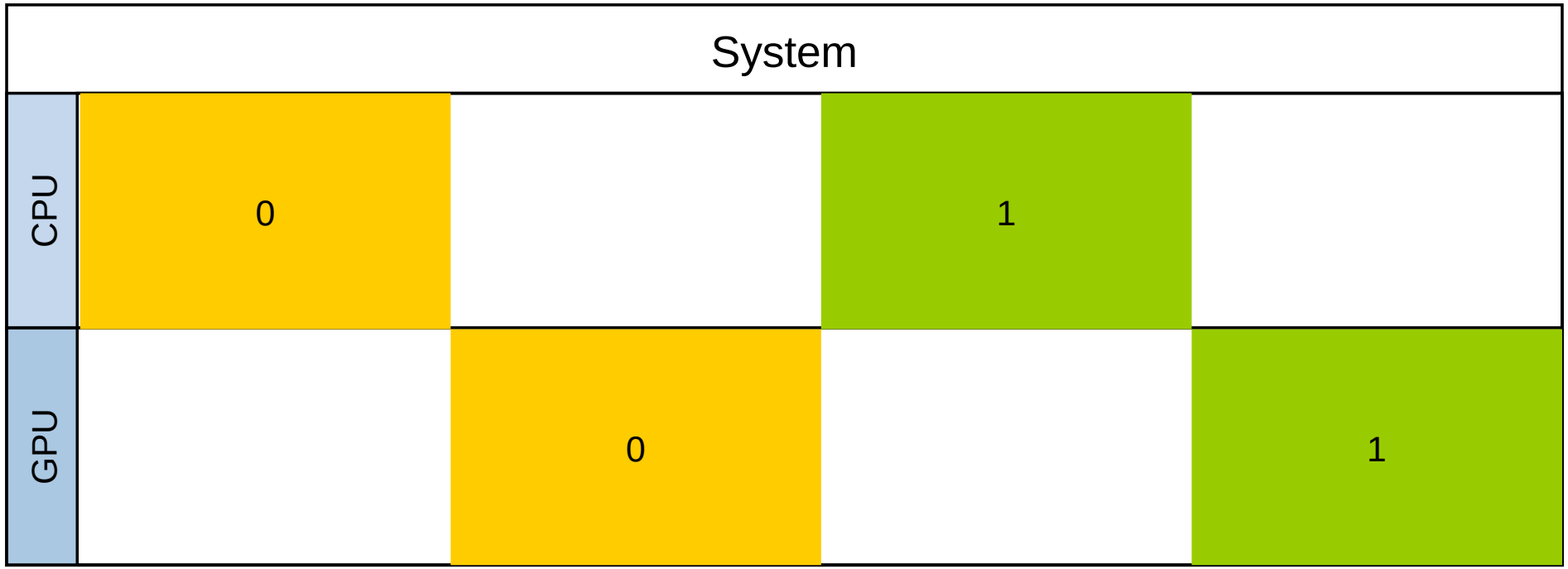
Introduction

- **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!

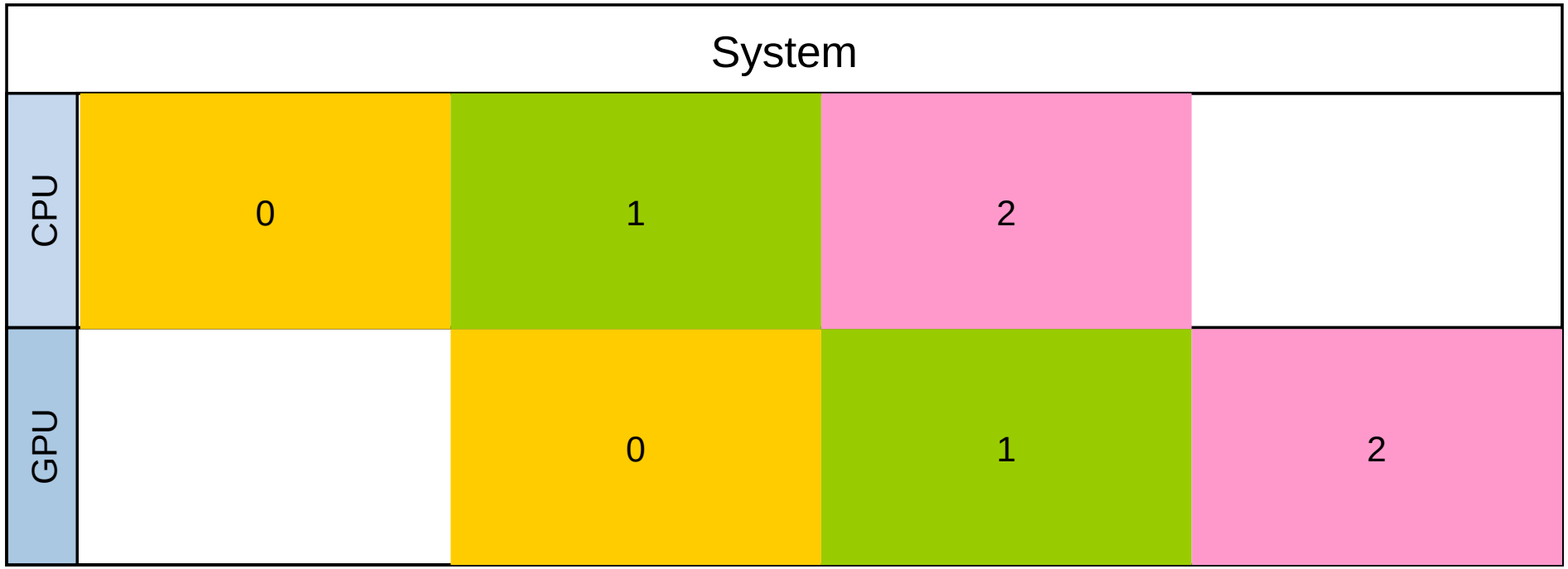
Keeping it fed

- **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.

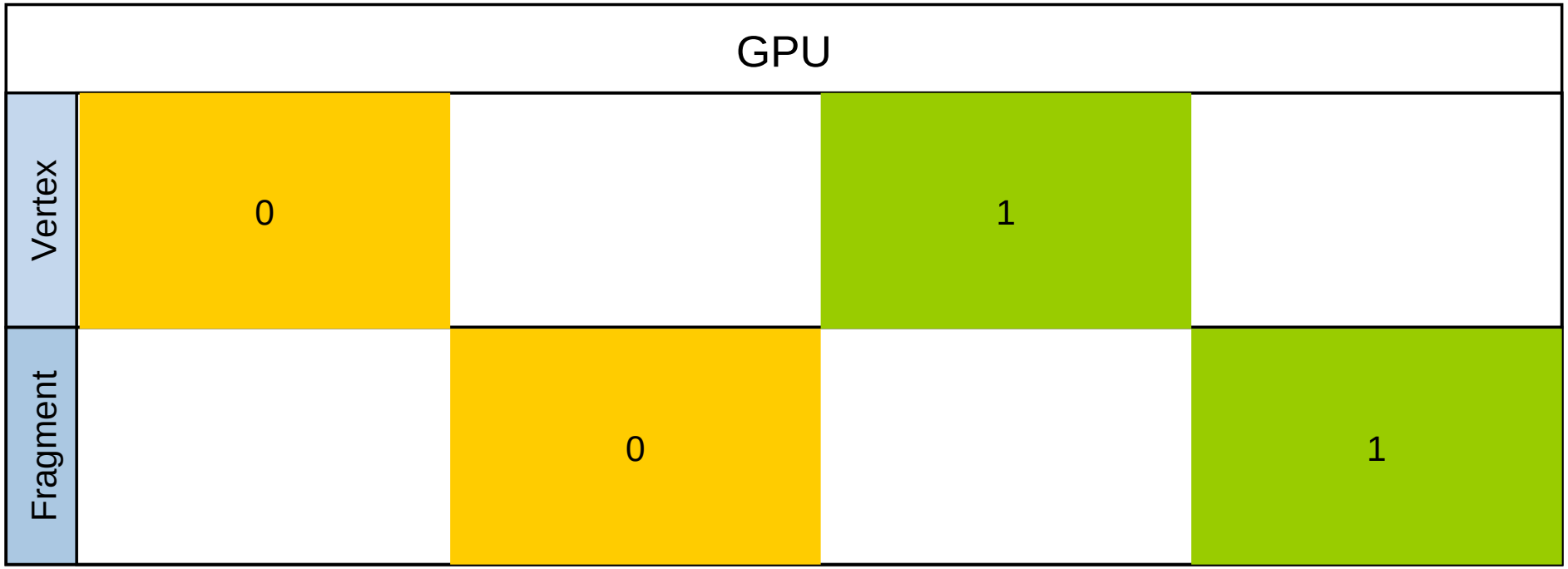
Keeping it fed



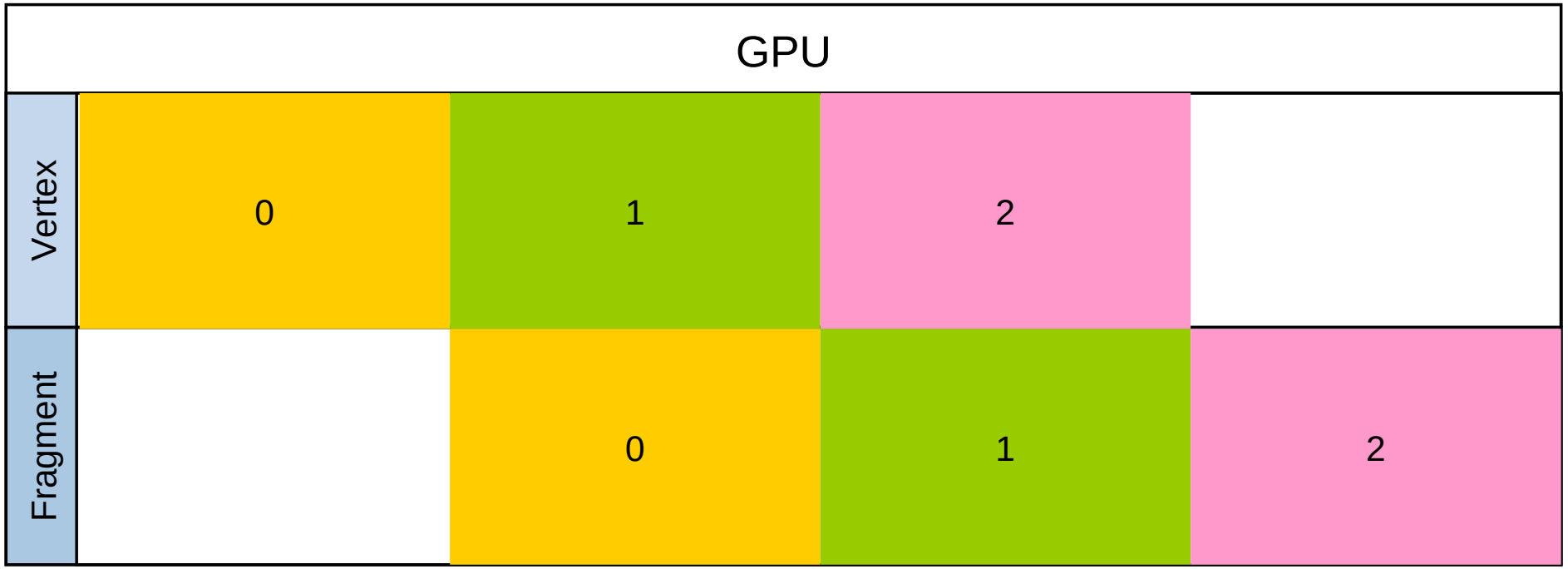
Keeping it fed



Keeping it fed



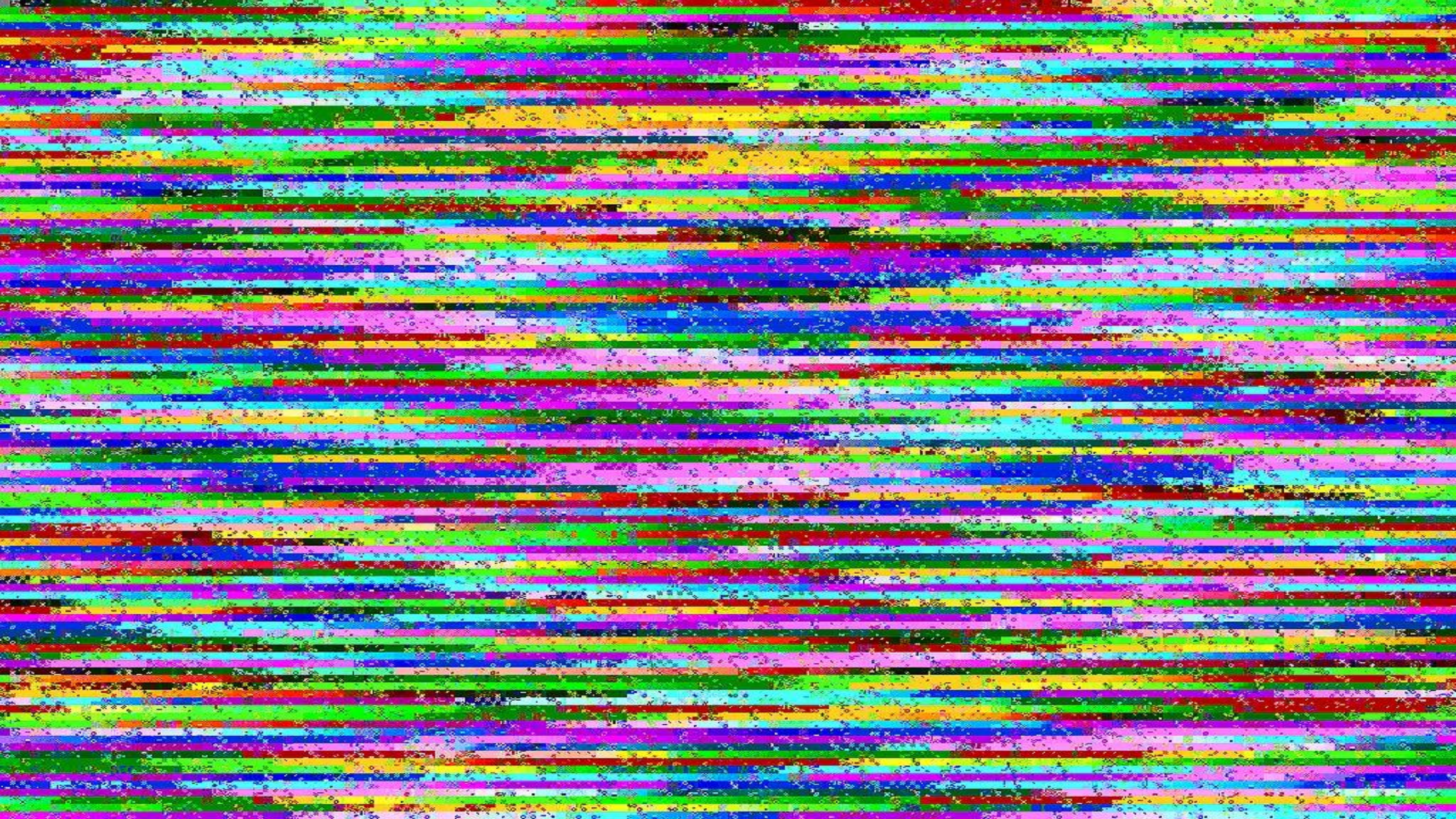
Keeping it fed



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


Tear Point #1 --->
Tear Point #2 --->



On to the serious bits…

Terminology

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

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

- **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

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

```
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);
```

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

Events

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

```
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

```
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?

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

VkResult vkResetEvent(
  VkDevice                    evice,
  VkEvent                     nt);

VkResult vkGetEve
  VkDevice
  VkEvent
```

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

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

Buffer Barriers

- **Buffer Barriers**
  - A single buffer range
  - Defines access stages
  - Defines queue ownership

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

```
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

```
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

```
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

```
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

```
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
externalDependency.srcSubpass = VK_SUBPASS_EXTERNAL;
externalDependency.srcStageMask = VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT;
externalDependency.dstStageMask = VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT;
externalDependency.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
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;
bufferBarrier.dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;
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

```
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)

```
// 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.

```
VkResult vkQueueSubmit(
  VkQueue                       queue,
  uint32_t                      submitCount,
  const VkSubmitInfo*           pSubmits,
  VkFence                       fence);

VkResult vkResetFences(
  VkDevice                      device,
  uint32_t                      fenceCount,
  const VkFence*                pFences);

VkResult vkGetFenceStatus(
  VkDevice                      device,
  VkFence                       fence);

VkResult vkWaitForFences(
  VkDevice                      device,
  uint32_t                      fenceCount,
  const VkFence*                pFences,
  VkBool32                      waitAll,
  uint64_t                      timeout);
```

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.

```
VkResult vkQueueSubmit(
  VkQueue                       queue,
  uint32_t                      submitCount,
  const VkSubmitInfo*           pSubmits,
  VkFence                       fence);

VkResult vkResetFences(
  VkDevice                      device,
  uint32_t                      fenceCount,
  const VkFence*                pFences);

VkResult vkGetFenceStatus(
  VkDevice                      device,
  VkFence                       fence);

VkResult vkWaitForFences(
  VkDevice                      device,
  uint32_t                      fenceCount,
  const VkFence*                pFences,
  VkBool32                      waitAll,
  uint64_t                      timeout);
```

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?

Break

Swapchains Unchained!

(What you need to know about Vulkan WSI)

Alon Or-bach, Chair, Vulkan System Integration Sub-Group – May 2016

@alonorbach (disclaimers apply!)

Intro to Vulkan Window System Integration

- **Explicit control for acquisition and presentation of images**
  - Designed to fit the Vulkan API and today's compositing window systems
- **Not all extensions are supported by every platform**
  - You MUST check and enable the extensions your app/engine uses!!!
- **Today's presentation should help you get presentation working**
  - Learn how to present through a swapchain
  - Overview of Vulkan objects used by the WSI extensions

WSI Jargon Buster

- **Platform**
  Our terminology for an OS / window system e.g. Android, Windows, Wayland, X11 via XCB
- **Presentation Engine**
  The platform's compositor or display engine
- **Application**
  Your app or game engine

How many WSI extensions are there?

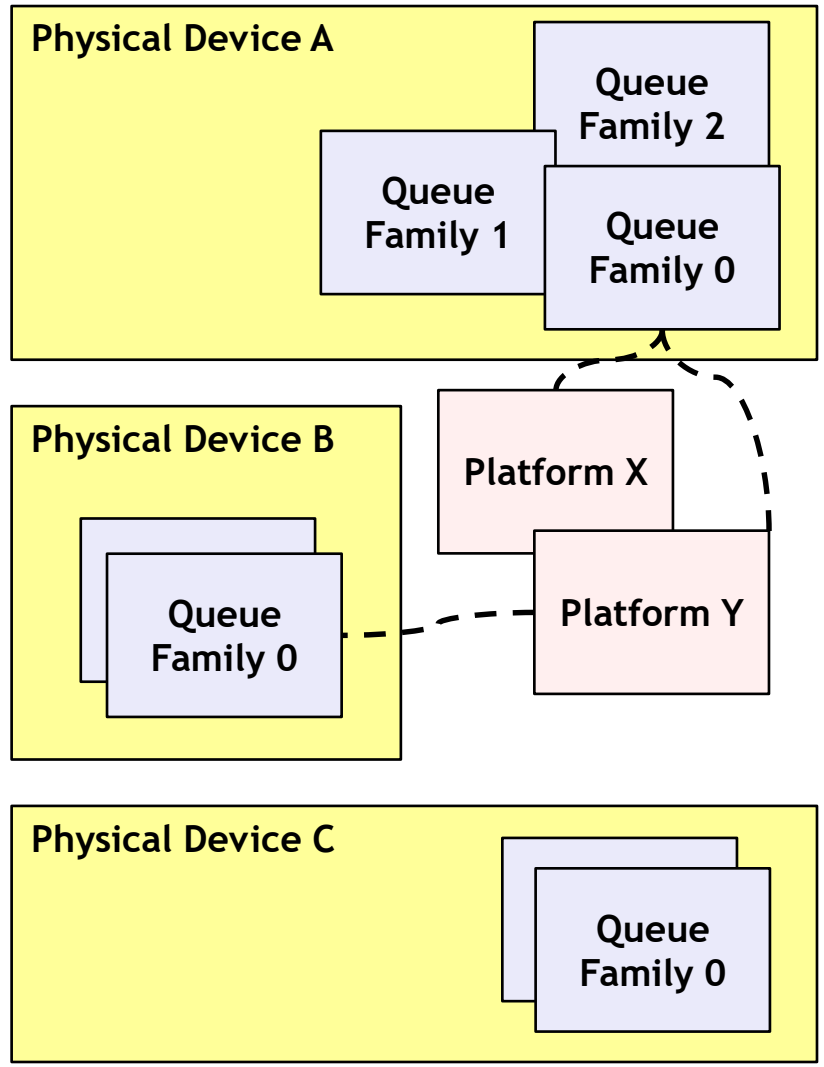
- **Two cross-platform instance extensions**
  - VK_KHR_surface
  - VK_KHR_display
- **Six (platform) instance extensions**
  - VK_KHR_android_surface
  - VK_KHR_mir_surface
  - VK_KHR_wayland_surface
  - VK_KHR_win32_surface
  - VK_KHR_xcb_surface
  - VK_KHR_xlib_surface
- **Two cross-platform device extensions**
  - VK_KHR_swapchain
  - VK_KHR_display_swapchain

Vulkan Surfaces

- **VkSurfaceKHR**
  - Vulkan's way to encapsulate a native window / surface

Unlike an EGLSurface, creating a Vulkan Surface doesn't mean you've got your render targets created …yet

- **Platform-independent surface queries**
  - Find out crucial information about your surface's properties
    - Such as format, transform, image usage
  - Some platforms provide additional queries
- **Presentation support is per queue family**
  - An implementation may support multiple platforms e.g. both xlib and xcb
  - Or may not support presentation at all



Vulkan Swapchains: VK_KHR_swapchain

- **Array of presentable images associated with a surface**
  - Application requests a minimum number of presentable images
  - Implementation creates at least that number
  - Implementation may have a limit
- **Upfront allocation of presentable images**
  - No allocation hitching at crucial moment
  - Pre-record fixed content command buffers
- **Present mode determines behavior**
  - FIFO support mandatory
  - Platforms can offer mailbox, immediate, FIFO relaxed

```
const VkSwapchainCreateInfoKHR createInfo =
{
 VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR, // sType
 NULL, // pNext
 0, // flags
 mySurface, // surface
 desiredNumberOfPresentableImages, // minImageCount
 surfaceFormat, // imageFormat
 surfaceColorSpace, // imageColorSpace
 myExtent, // imageExtent
 1, // imageArrayLayers
 VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, // imageUsage
 VK_SHARING_MODE_EXCLUSIVE, // imageSharingMode
 0, // queueFamilyIndexCount
 NULL, // pQueueFamilyIndices
 surfaceProperties.currentTransform, // preTransform
 VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR, // compositeAlpha
 swapchainPresentMode, // presentMode
 VK_TRUE, // clipped
 VK_NULL_HANDLE // oldSwapchain
};
```

FIFO is like eglSwapInterval = 1
Mailbox/Immediate is like eglSwapInterval 0
FIFO relaxed is like EXT_swap_control_tear

Vulkan Swapchains: They're good!

- **Application knows which image within a swapchain it is presenting**
  - Content of image preserved between presents
- **Application is responsible for explicitly recreating swapchains - no surprises**
  - Platform informs app if current swapchain
    - Suboptimal: e.g. after window resize, swapchain still usable for present via image scaling
    - Surface Lost: swapchain no longer usable for present
  - Application is responsible to create a new swapchain

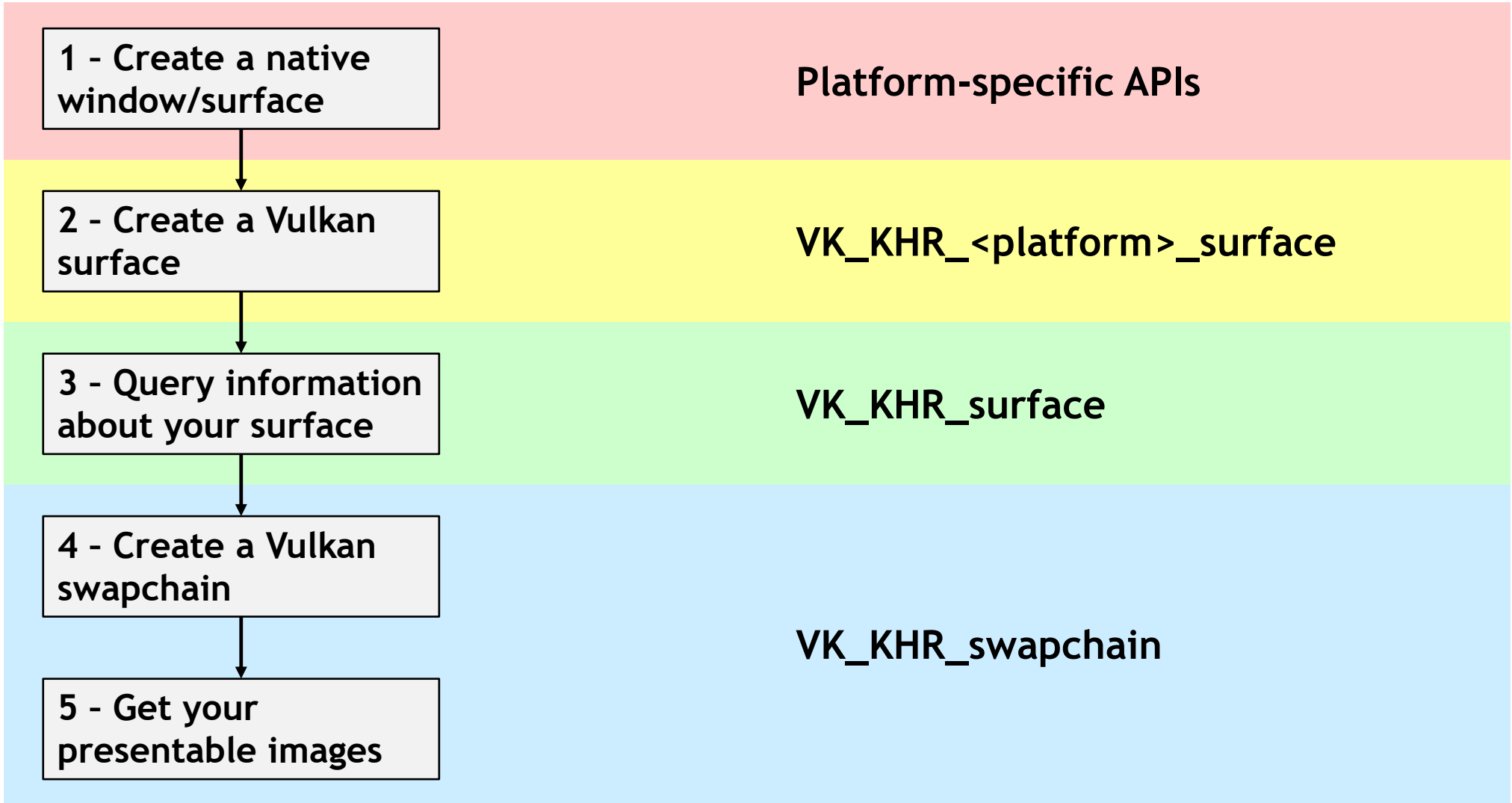
In EGL, the EGLSurface may be resized by the platform after an eglSwapBuffers call.
Vulkan requires the application to intervene

Vulkan Swapchains: They're jolly good!

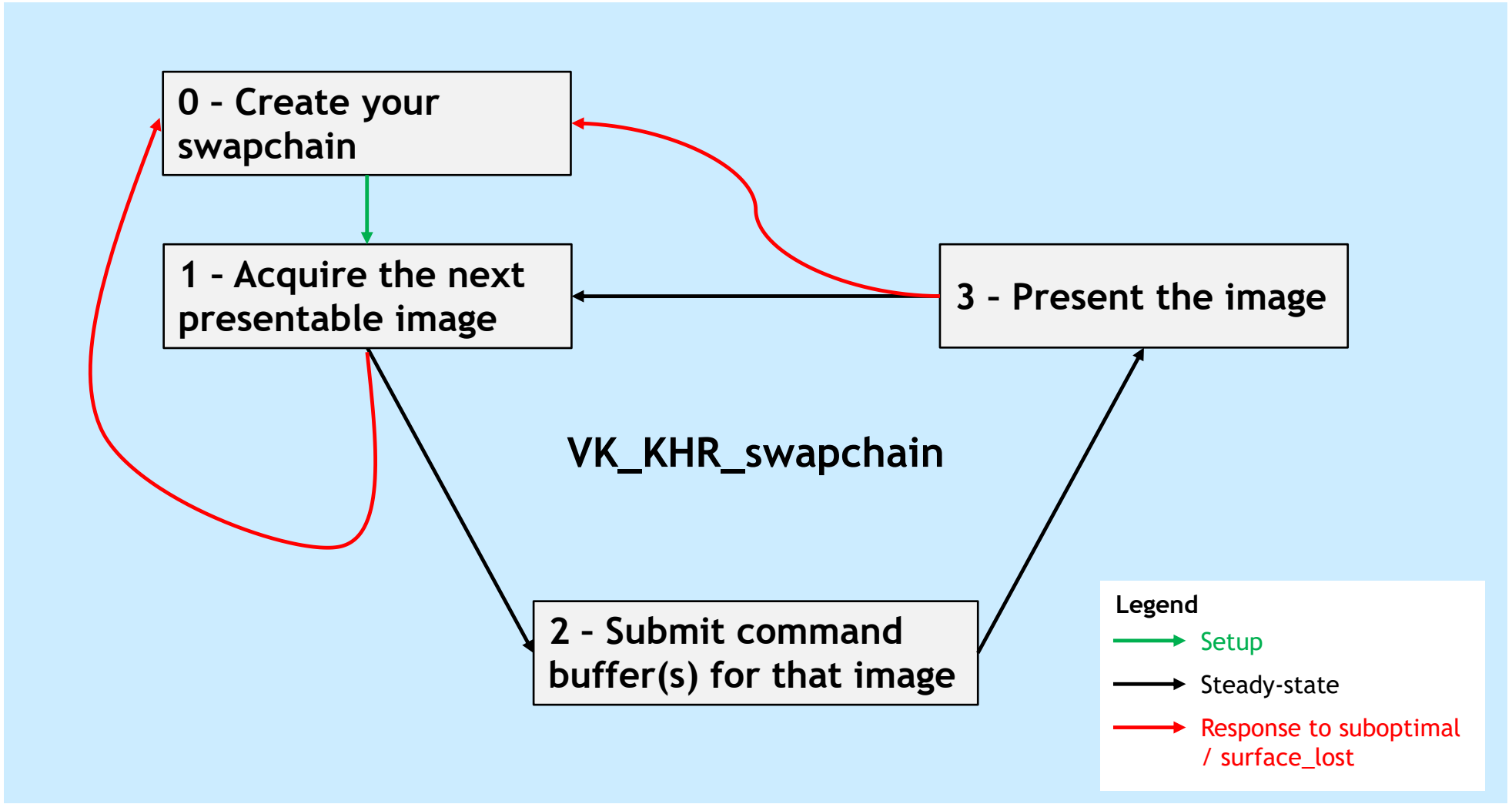
- **Presenting and acquiring are separate operations**
  - No need to submit a new image to acquire another one, unless presentation engine cannot release it
- **Application must only modify presentable images it has acquired**
- **Presentation engine must only display presentable images that have been presented!**

In EGL, calling eglSwapBuffers both presents the current back buffer **and** acquires a new one
Vulkan splits this up into separate operations

Steps to setup your presentable images

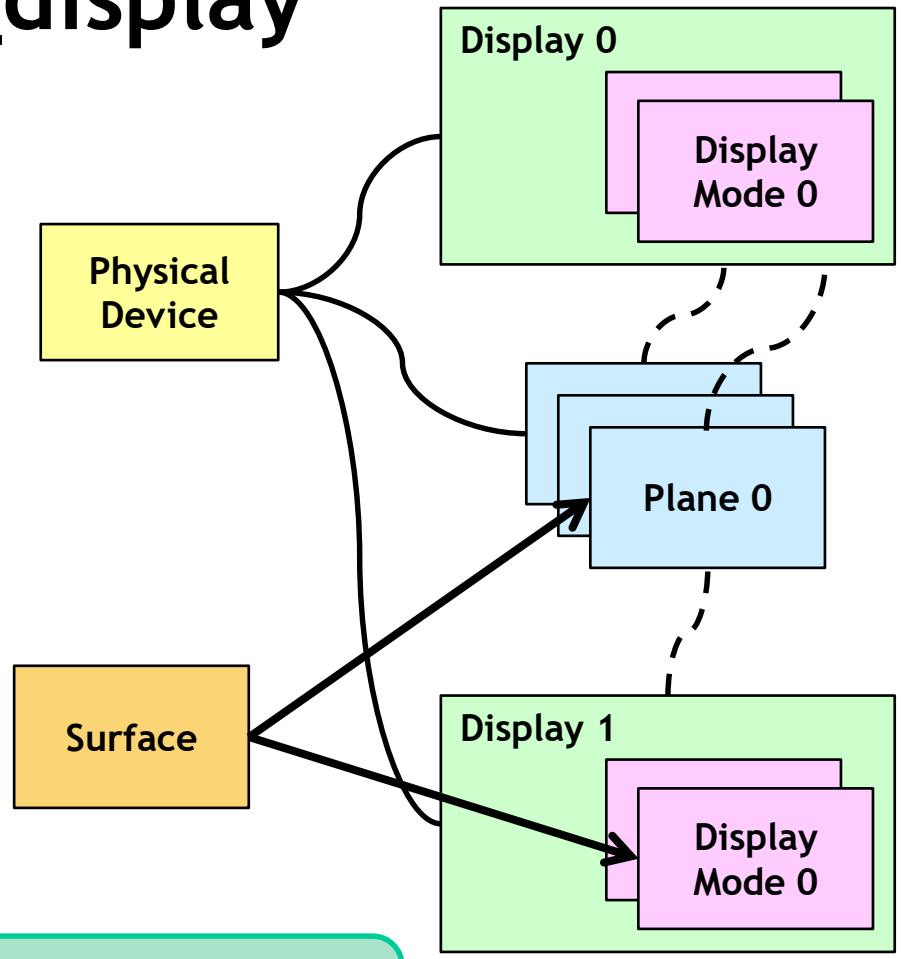


Vulkan Frame Loop - as easy as 1-2-3!



Vulkan Displays: VK_KHR_display

- **Vulkan's way to discover display devices (screens, panels) outside a window system**
  - Reminder: Not supported on all platforms
- **Defines VkDisplayKHR and VkDisplayModeKHR objects**
  - Represent the display devices and the modes they support connected to a VkPhysicalDevice
  - Determine if a display supports multiple planes that are blended together
- **Enables creation of a VkSurfaceKHR to represent a display plane**



A Vulkan display represents an actual display!
(Whereas an EGLDisplay is actually just a connection to a driver – like a Vulkan Device)

VK_KHR_display_swapchain

- **Extends the information provided at vkQueuePresentKHR**
  - What region to present from the swapchain image
  - What region to present to on the display
  - Whether the display should persist the image
- **Adds ability to create a shared swapchain**
  - Swapchain that takes multiple VkSwapchainCreateInfoKHR structs
  - Allows multiple displays to be presented to simultaneously
  - No guarantee that presents are atomic ...presently!

Any question?

alon.orbach@samsung.com
@alonorbach

Moving To Vulkan

Asynchronous Compute

Chris Hebert, Dev Tech Software Engineer, Professional Visualization

Who am I?

Chris Hebert

@chrisjhebert

Dev Tech Software Engineer- Pro Vis

20 years in the industry

Joined NVIDIA in March 2015.

Real time graphics makes me happy

I also like helicopters

Chris Hebert - Circa 1974

Agenda

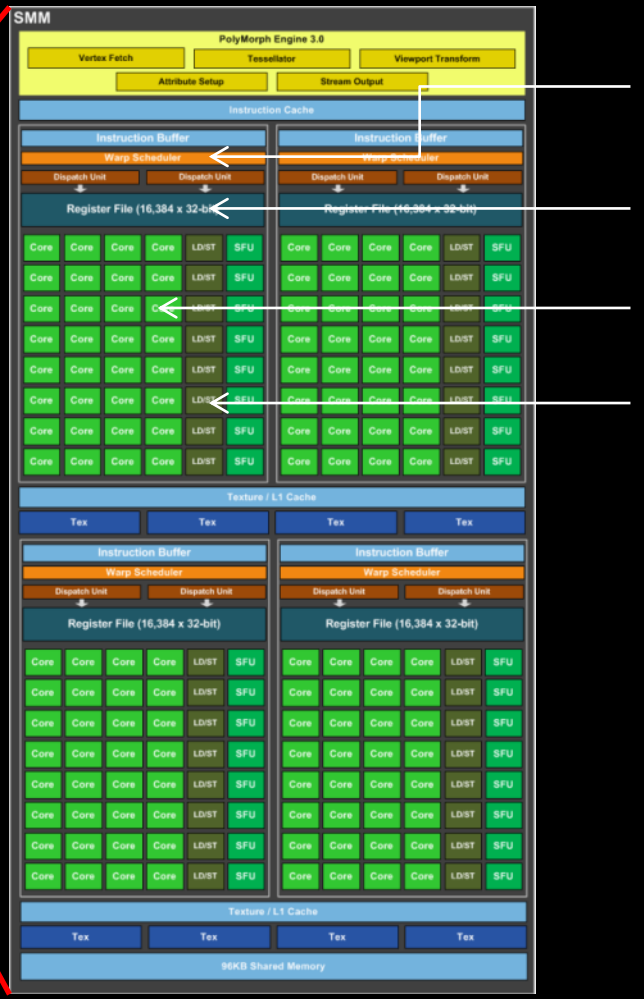
- Some Context
- Sharing The Load
- Pipeline Barriers

Some Context

GPU Architecture

In a nutshell

NVIDIA Maxwell 2



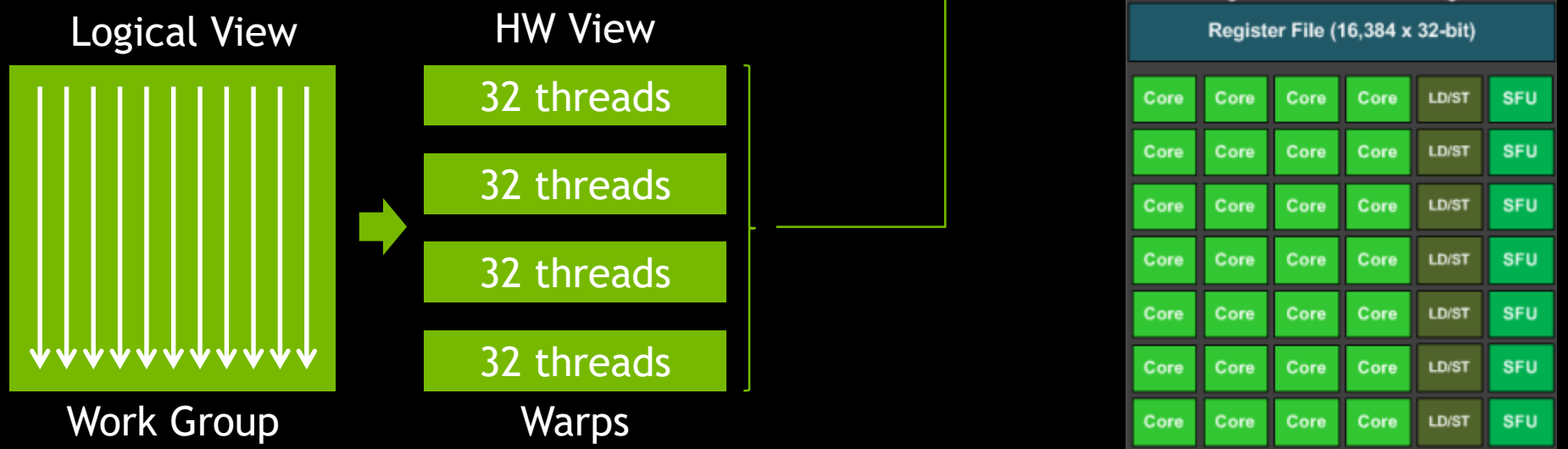
Register File

Core

Load Store Unit

Execution Model

Thread Hierarchies



Resource Partitioning

Resources Are Limited

Key resources impacting local execution:

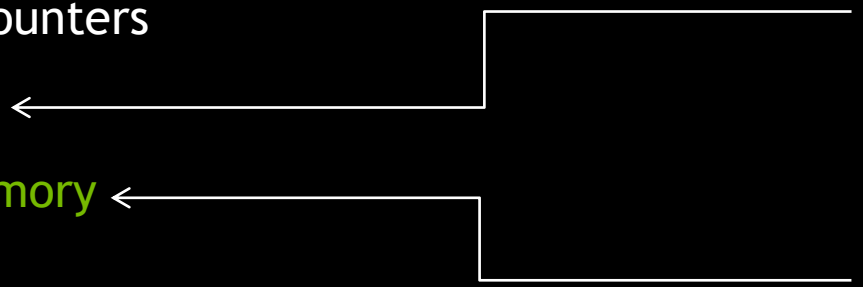
- Program Counters
- Registers
- Shared Memory

Resource Partitioning

Resources Are Limited

Key resources impacting local execution:

- Program Counters
- Registers
- Shared Memory



Partitioned amongst threads

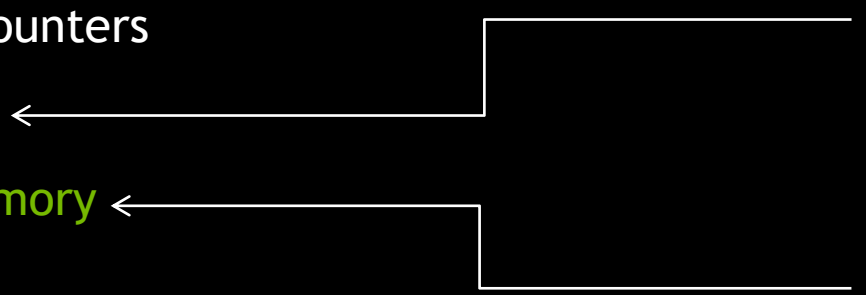
Partitioned amongst work groups

Resource Partitioning

Resources Are Limited

Key resources impacting local execution:

- Program Counters
- Registers
- Shared Memory



Partitioned amongst threads

Partitioned amongst work groups

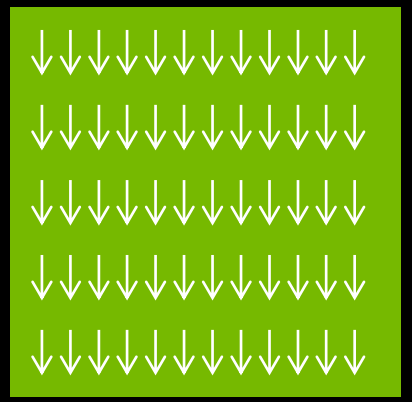
e.g. GTX 980 ti
64k 32bit registers per SM
96kb shared memory per SM

Resource Partitioning

Registers

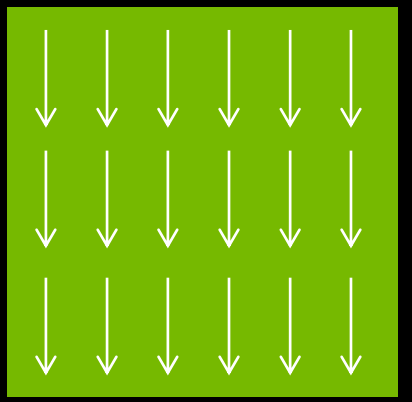
The more registers used by a kernel means few resident warps on the SM

Fewer Registers



More Threads

More Registers



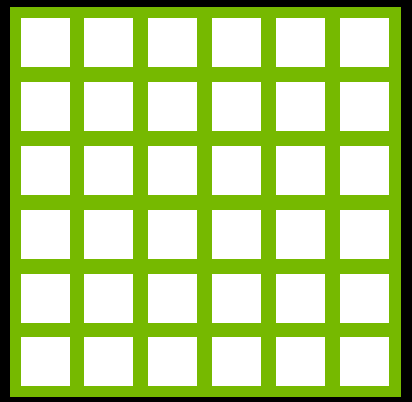
Fewer Threads

Resource Partitioning

Shared Memory

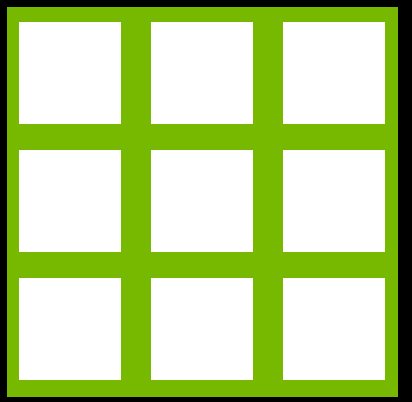
The more shared memory used by a work group means fewer work groups on the SM

Less SMEM



More Groups

More SMEM



Fewer Groups

Keeping It Moving
Occupancy

- Some small kernels may have low occupancy
  - Depending on the algorithm
- Compute resources are limited
  - Shared across threads or work groups on a per SM basis
- Warps stall when they have to wait for resources
- This latency can be hidden
  - If there are other warps ready to execute.

Keeping It Moving

Occupancy – Simple Theoretical Example

- Simple kernel that updates positions of 20480 particles
  - 1 FMAD - ~20 cycles (instruction latency)
  - 20480 particles = 640 warps
  - To hide this latency, according to Littles Law
    - Required Warps = Latency x Throughput
    - Throughput should be 32 threads * 16 sms = 512 to keep GPU busy
    - Required warps is 20*512 = 10240
    - ….oh….

Keeping It Moving

Occupancy – Simple Theoretical Example

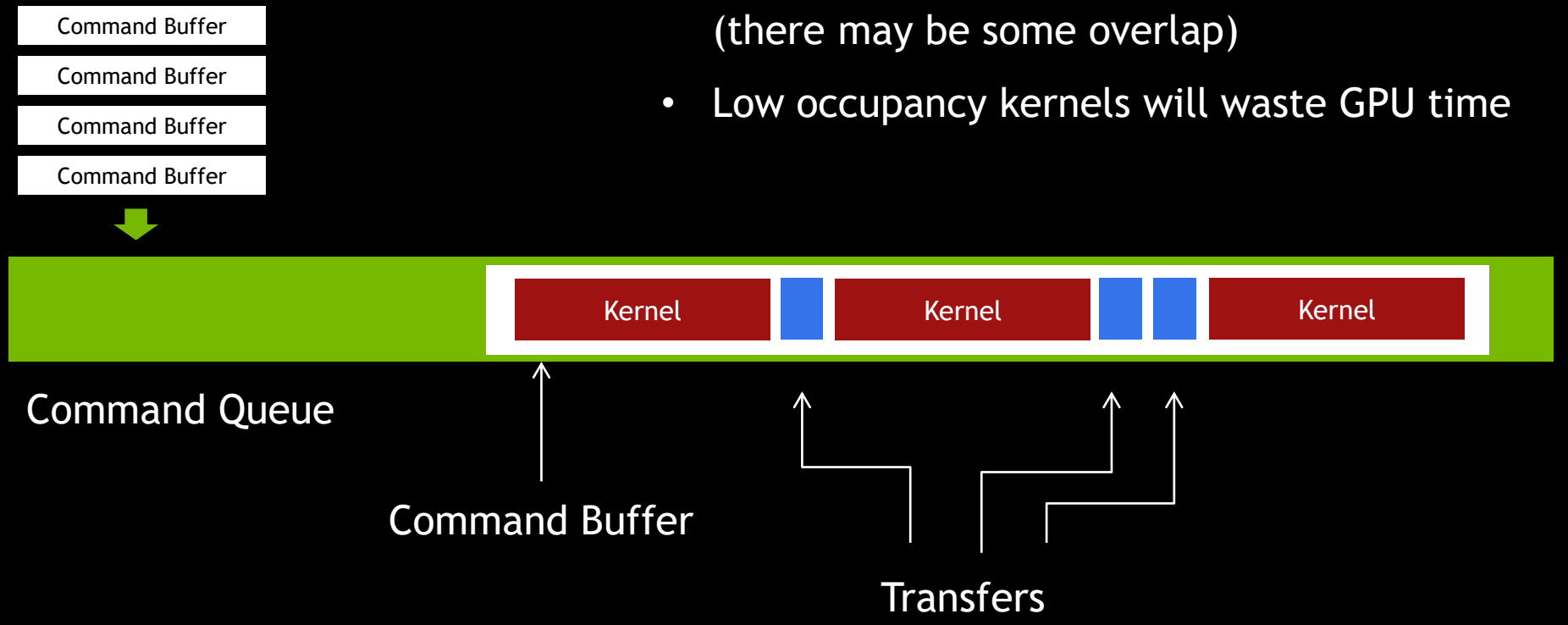
- Simple kernel that updates positions of 20480 particles
  - 1 FMAD - ~20 cycles (instruction latency)
  - 20480 particles = 640 warps
  - To hide this latency, according to Littles Law – But only on 1 SM..
    - Required Warps = Latency x Throughput
    - Throughput should be 32 threads * 1 sm = 32 to keep GPU busy
    - Required warps is 20*32 = 640
    - And we theoretically have 15 SMs to use for other stuff.

Queuing It Up

Working with 1 Queue

- Scheduler will distribute work across all SMs
- kernels execute in sequence

  (there may be some overlap)
- Low occupancy kernels will waste GPU time



Sharing The Load

Queuing It Up

Working with N Queues

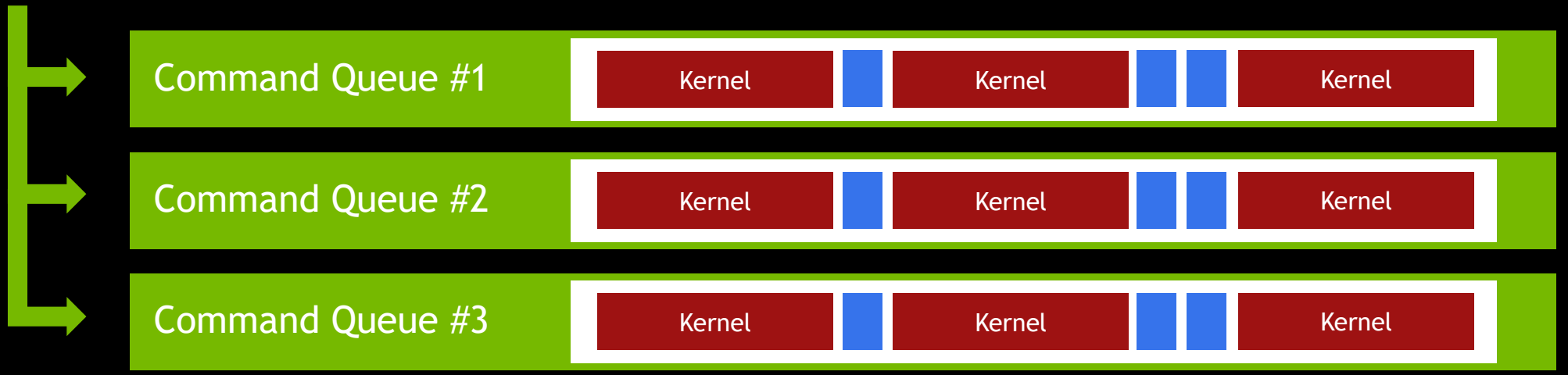
Command Buffer

Command Buffer

Command Buffer

Command Buffer

- NVIDIA hardware gives you 16 all powerful queues
- 1 Queue family that supports all operations
- 16 queues available for use



Queuing It Up

Working with N Queues

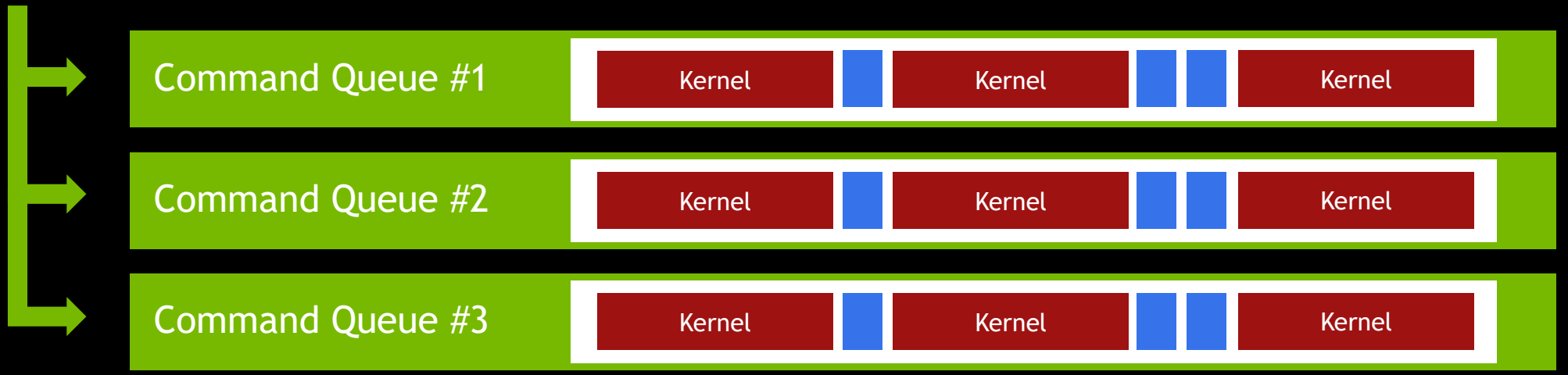
Command Buffer

Command Buffer

Command Buffer

Command Buffer

- Application decides which queues for which kernels
- Load balance for best performance
- Profile (Nsight) to gain insights



Queuing It Up

Compute and Graphics In Harmony

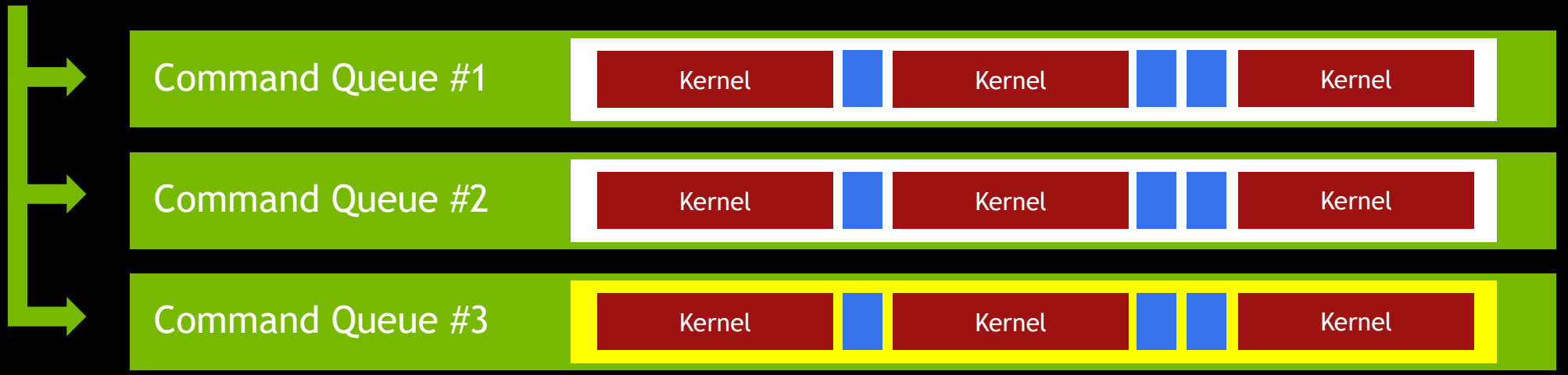
- Some hardware can even run compute and graphics work concurrently
- Needs fast context switching and at high granularity (not just at draw commands)
- Simple Graphics work tends to have high occupancy
- Complex graphics work can reduce occupancy
- Profile for performance insights

Queuing It Up

Compute and Graphics In Harmony

Compute Cmd Buffer
Compute Cmd Buffer
Graphics Cmd Buffer
Compute Cmd Buffer

- Profile to understand occupancy of both graphics and compute workloads
- Queues can support both compute and graphics



An Example

Compute and Graphics In Harmony

Click here to view this video

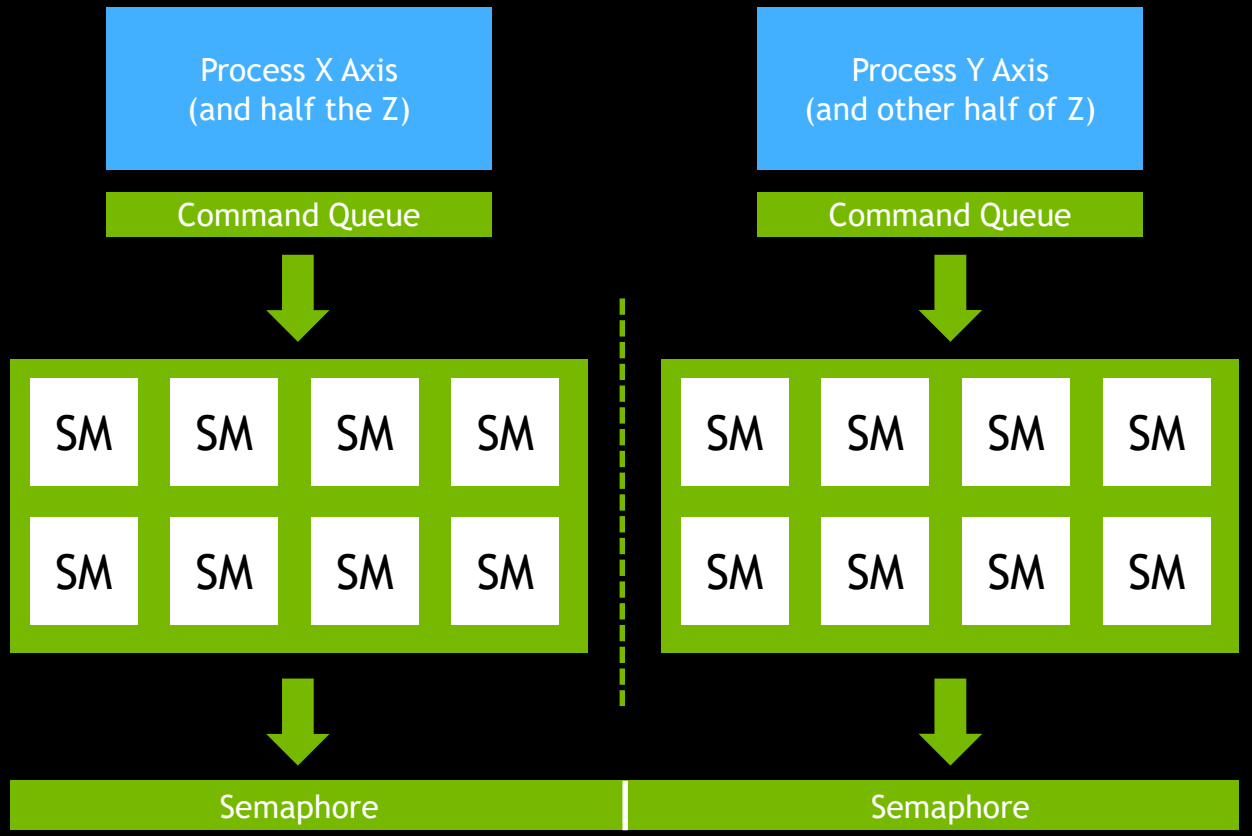
Free Surface Navier Stokes Solver

- 11 Compute Kernels
- 4 Shaders

- The output of each kernel is the input to the next
- Some kernels have very low occupancy
- Still opportunities for concurrency with compute

An Example

Many discretized operations are separable



Examples

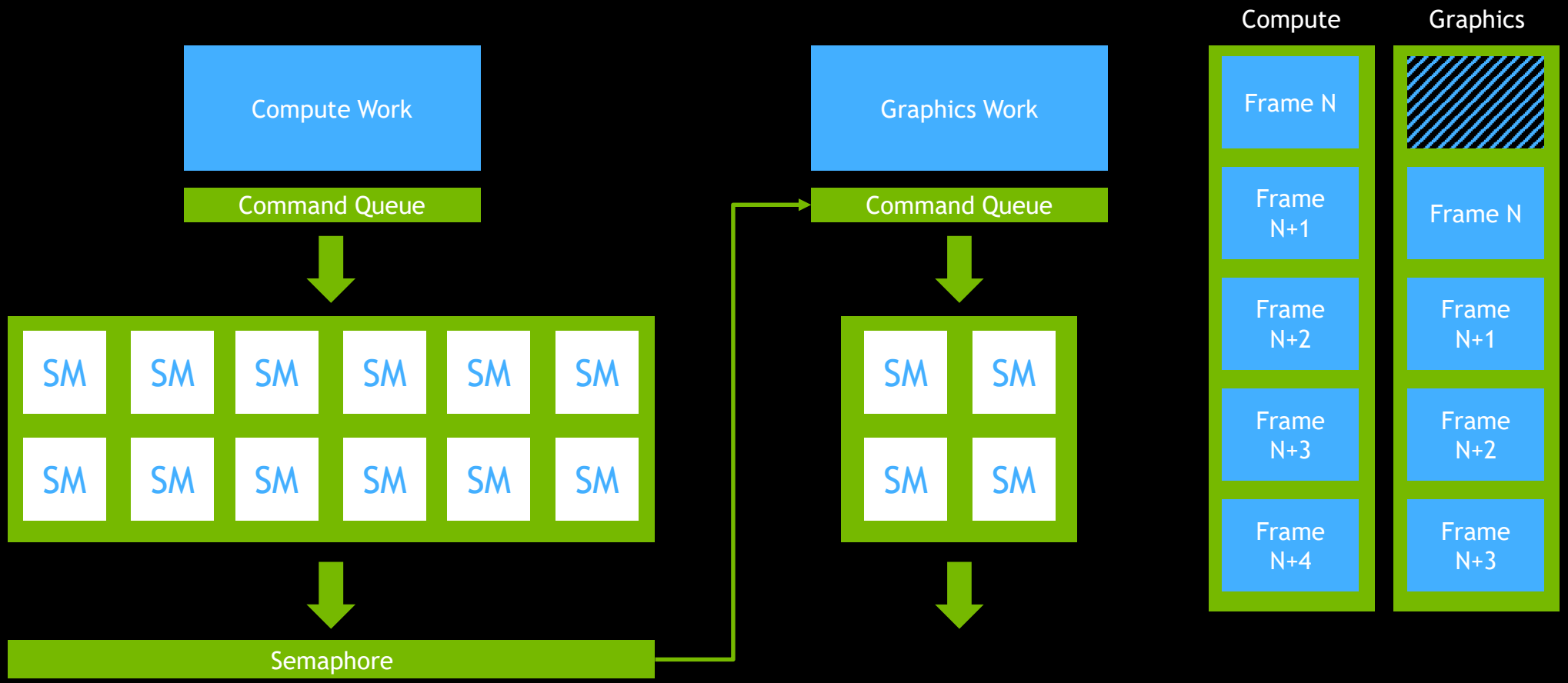
- Fluid Sims
- Gaussian Blurs
- Convolution Kernels

Driver handles dispatching groups

Use semaphores to synchronize

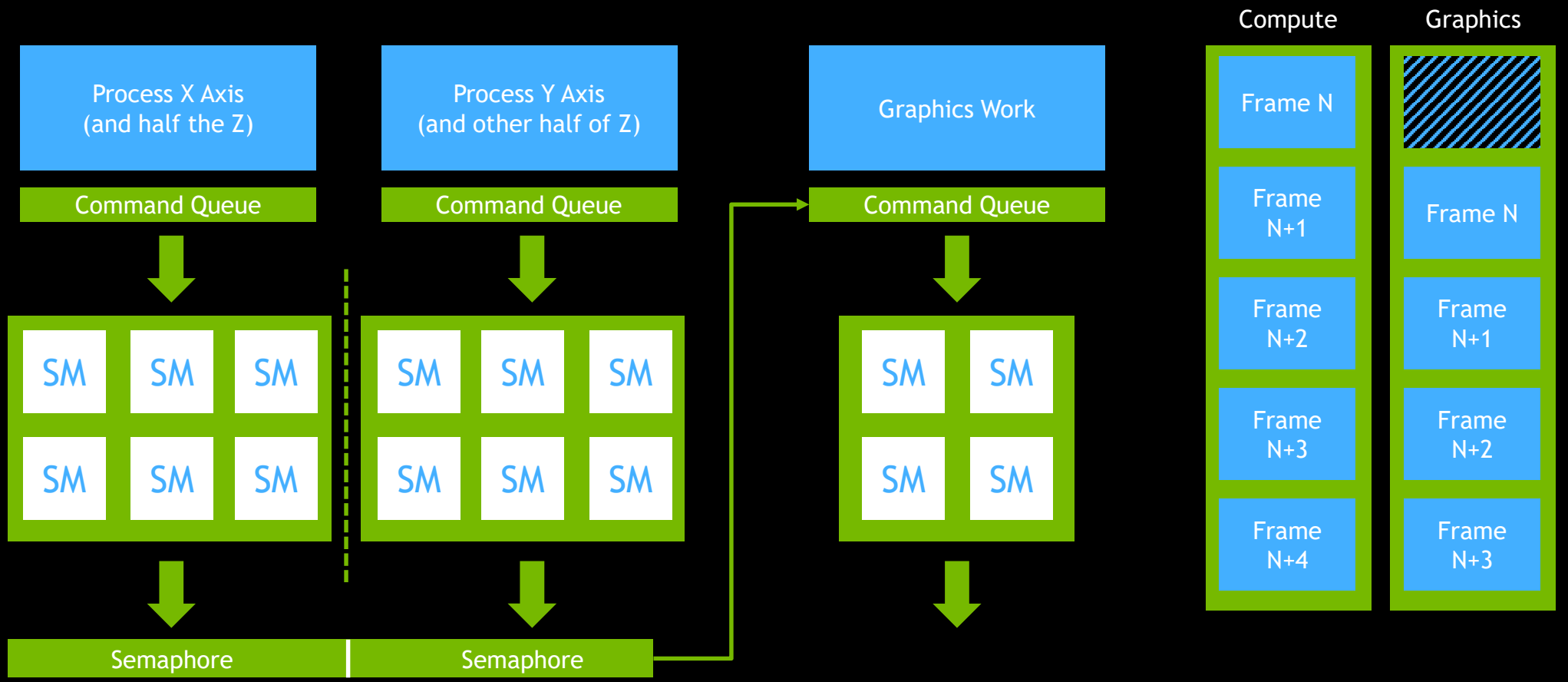
An Example

Compute and graphics run concurrently



An Example

Putting it all together

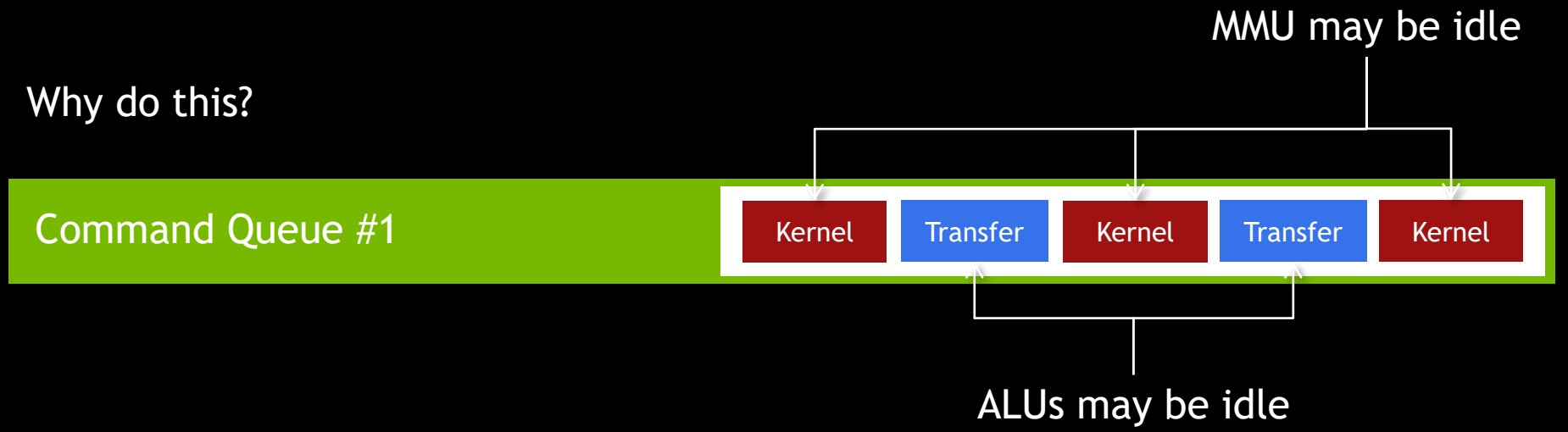


Memory Transfers

More opportunity for concurrency

- Memory transfers are handle by MMU
- Can run concurrently with Kernels
  - As long as the current kernel isnt using the memory

Why do this?

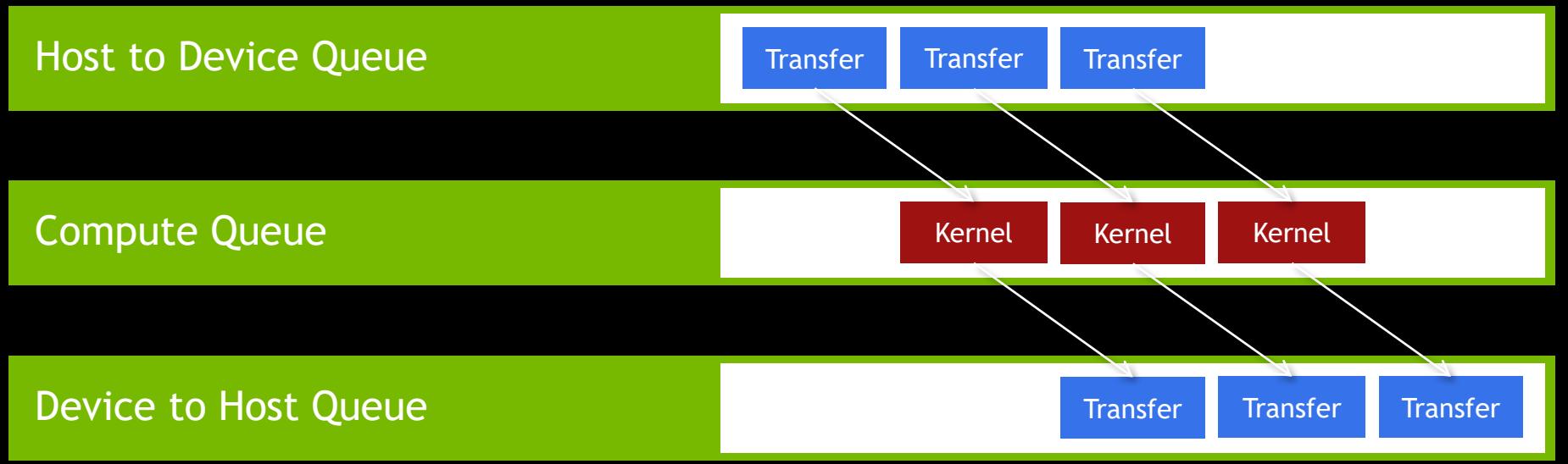


Memory Transfers

More opportunity for concurrency

When you can do this
- DtoH and HtoD transfers can run concurrently

Examples
- Large image processing
- Video processing



Conclusion

Takeaways

There is more than 1 queue available

Keep registers and shared memory to a minimum

Low occupancy leads to an under utilized GPU

Maximize GPU utilization by running kernels concurrently

Profile to understand the occupancy profiles of kernels and shaders

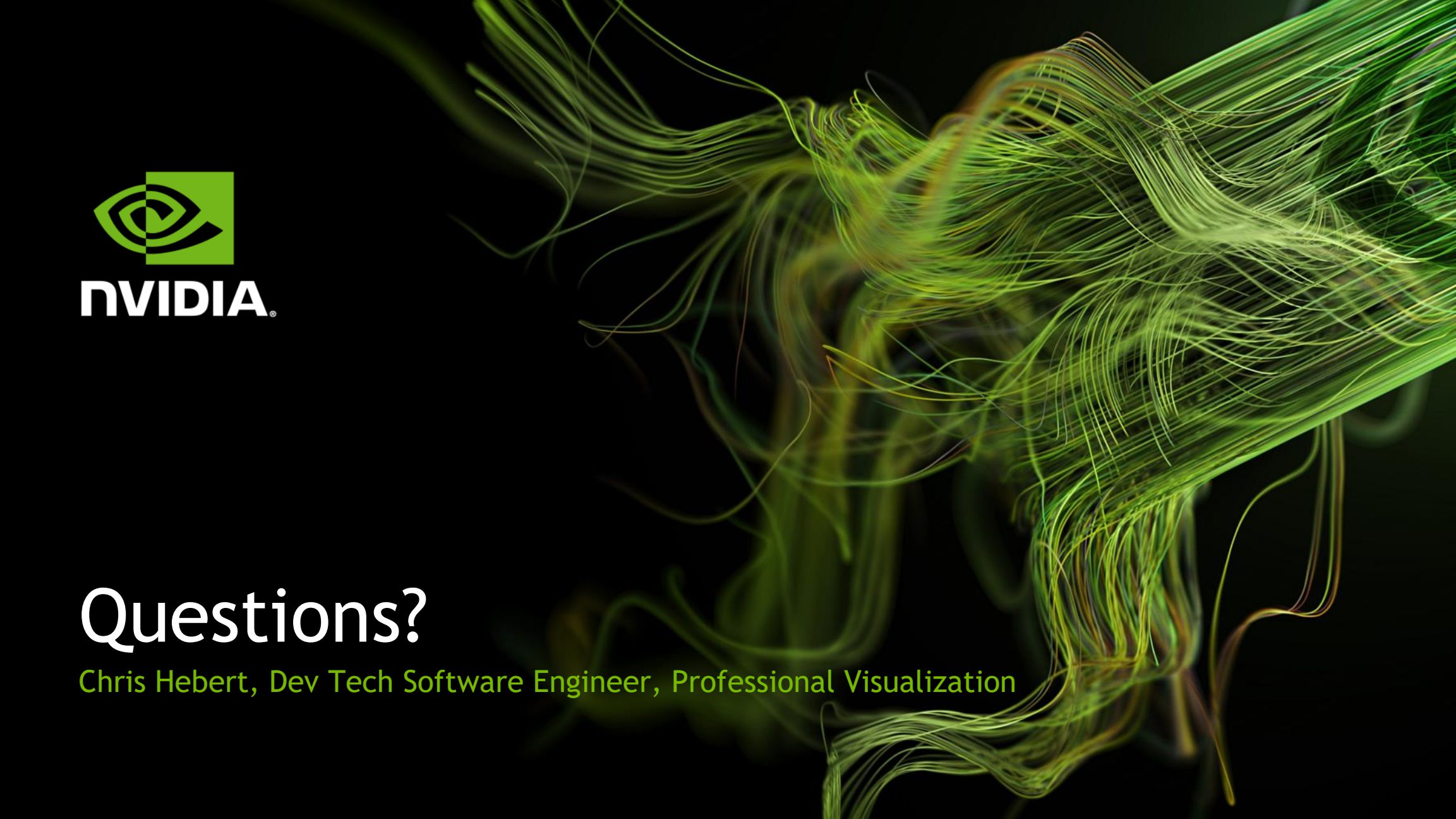
Some hardware can run kernels AND shaders concurrently

Use Semaphores to synchronize between queues

Be sensible at the beer festival

Thank You

Enjoy Vulkan!!



Questions?

Chris Hebert, Dev Tech Software Engineer, Professional Visualization

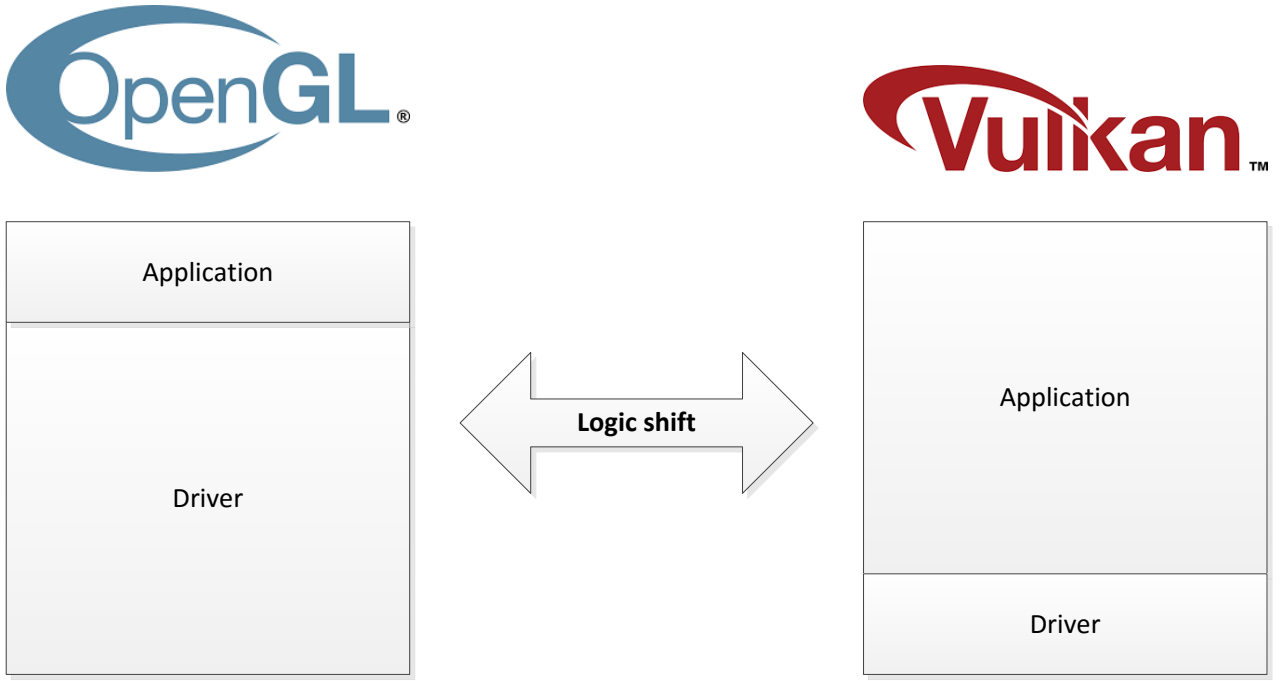
Porting to Vulkan

Hans-Kristian Arntzen
Engineer, ARM
(Credit for slides: Marius Bjørge)

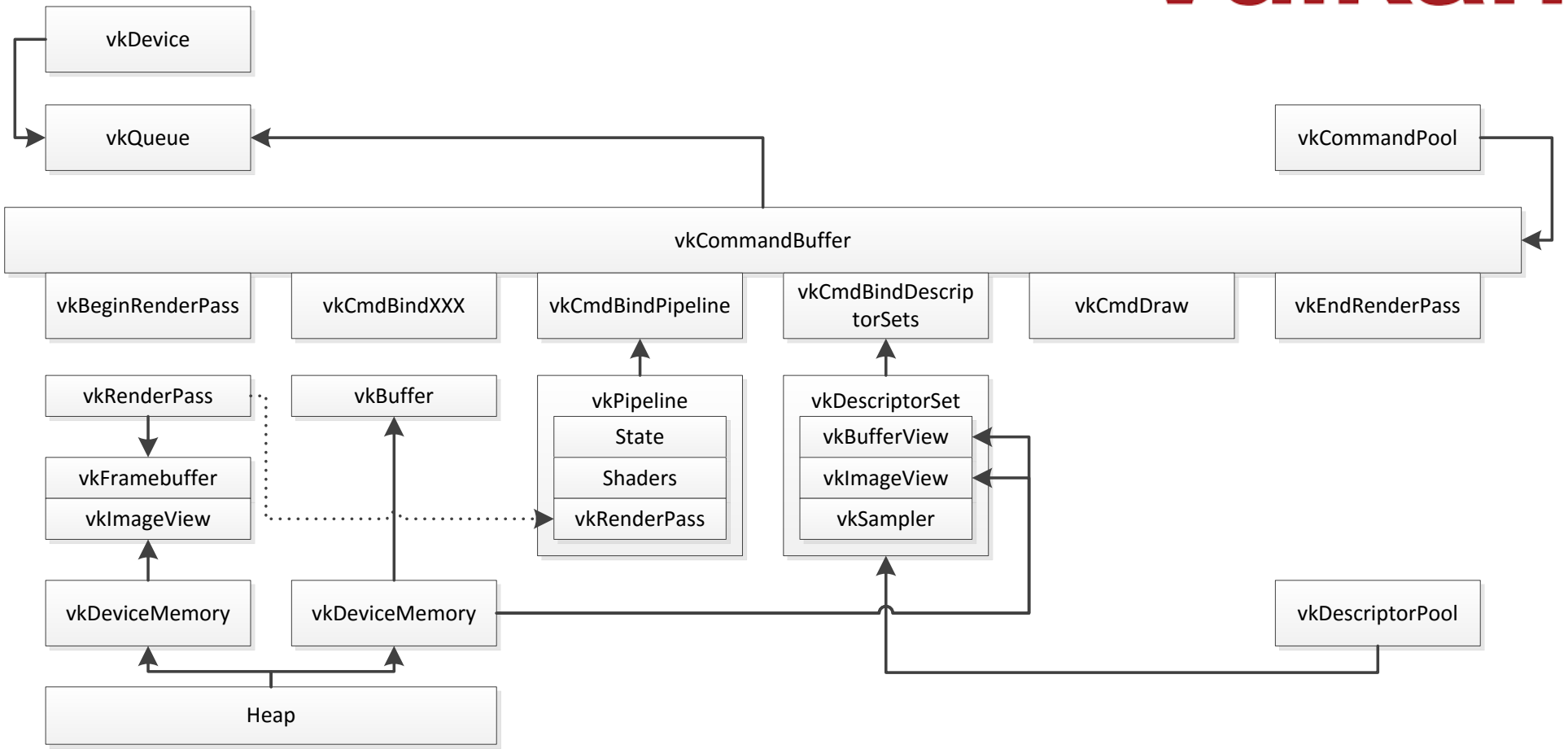
Agenda

- **API flashback**
- **Engine design**
  - Command buffers
  - Pipelines
  - Render passes
  - Memory management

API Flashback



API Flashback



Porting from OpenGL to Vulkan?

- **Most graphics engines today are designed around the principles of implicit driver behaviour**
  - A direct port to Vulkan won't necessarily give you a lot of benefits

- **Approach it differently**
  - Re-design for Vulkan, and then port that to OpenGL

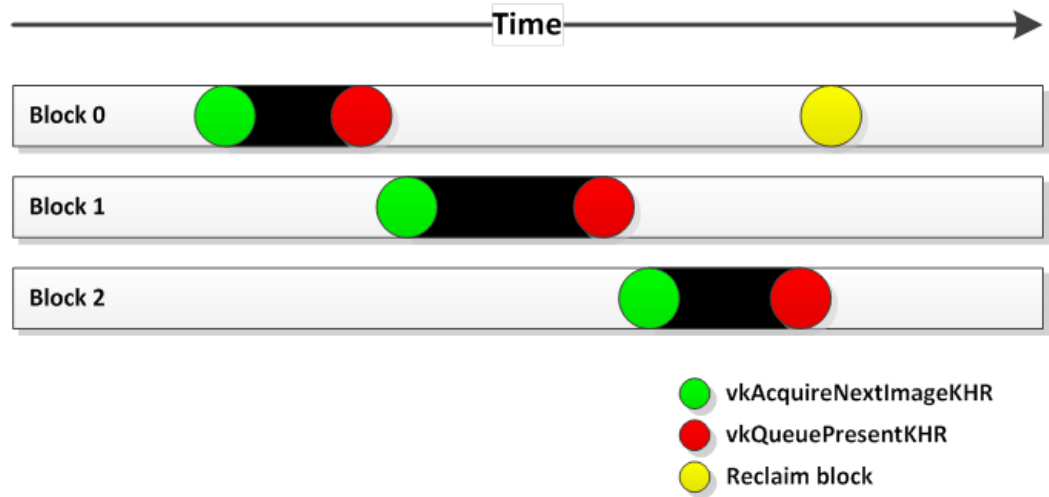
Allocating Memory

- **Memory is first allocated and then bound to Vulkan objects**
  - Different Vulkan objects may have different memory requirements
  - Allows for aliasing memory across different Vulkan objects
- **Driver does no ref counting of any objects in Vulkan**
  - Cannot free memory until you are sure it is never going to be used again
  - Also applies to API handles!

- **Most of the memory allocated during run-time is transient**
  - Allocate, write and use in the same frame
  - Block based memory allocator

Block Based Memory Allocator

- Relaxes memory reference counting
- Only entire blocks are freed/recycled
- Sub-allocations take refcount on block

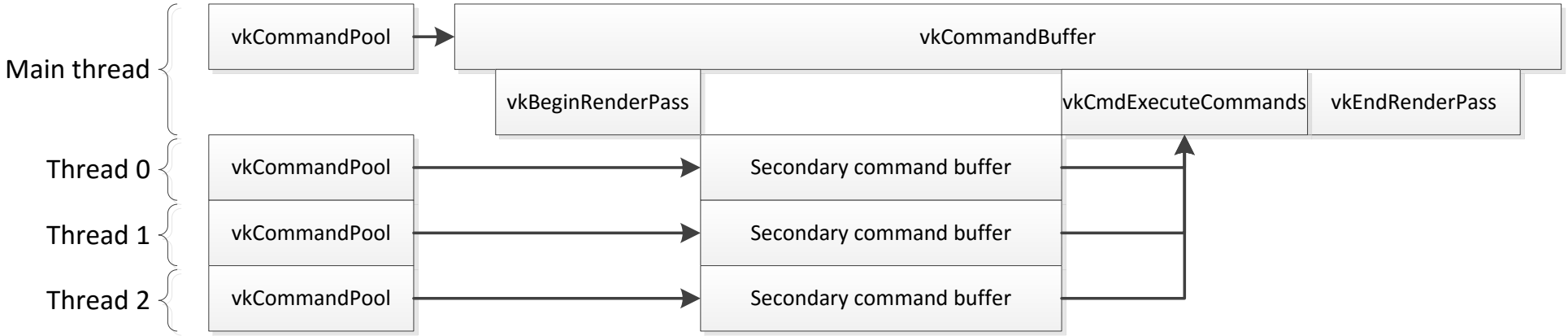


Command Buffers

- **Request command buffers on the fly**
  - Allocated using ONE_TIME_SUBMIT_BIT
  - Recycled

- **Separate command pools per**
  - Thread
  - Frame
  - Primary/secondary

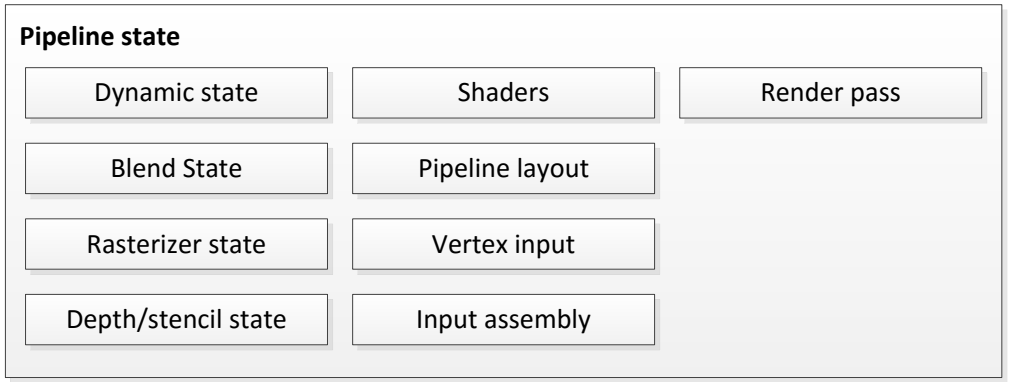
Secondary Command Buffers



Shaders

- **Standardize on SPIR-V binary shaders**
- **Extensively use the Khronos SPIRV-Cross library**
  - Cross compiling back to GLSL
  - Provides shader reflection for
    - Vertex attributes
    - Subpass attachments
    - Pipeline layouts
    - Push constants

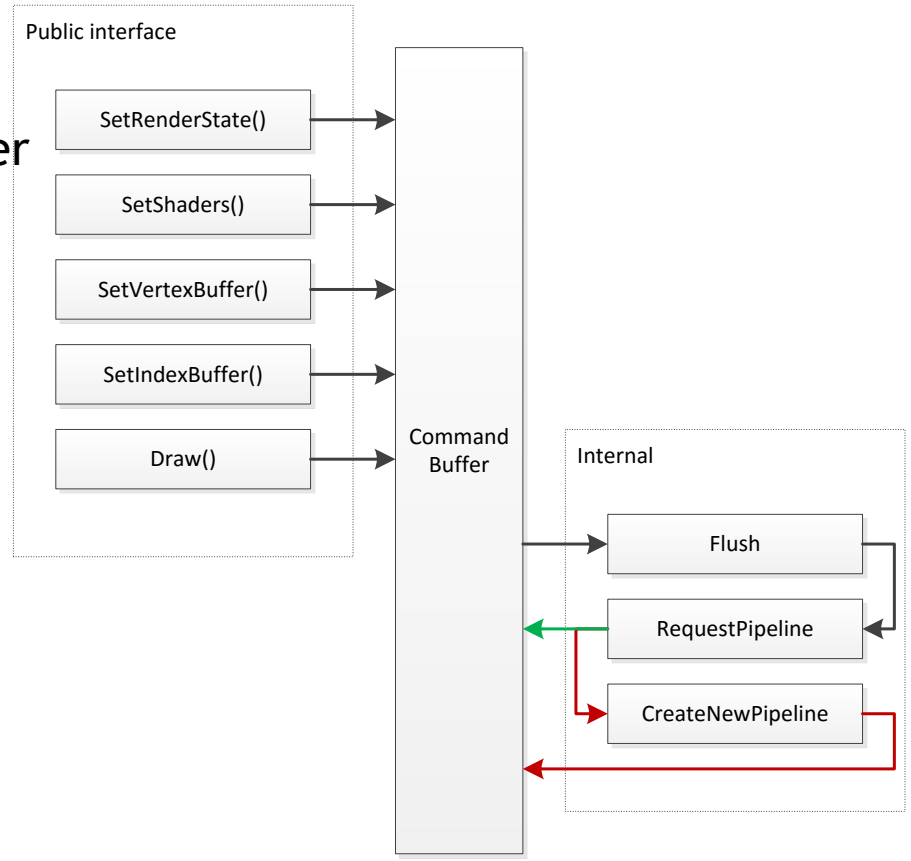
Pipelines



Pipelines

- **Not trivial to create all required pipeline state objects upfront**

- **Our approach:**
  - Keep track of all pipeline state per command buffer
  - Flush pipeline creation when required
    - In our case this is implemented as an async operation



Pipelines

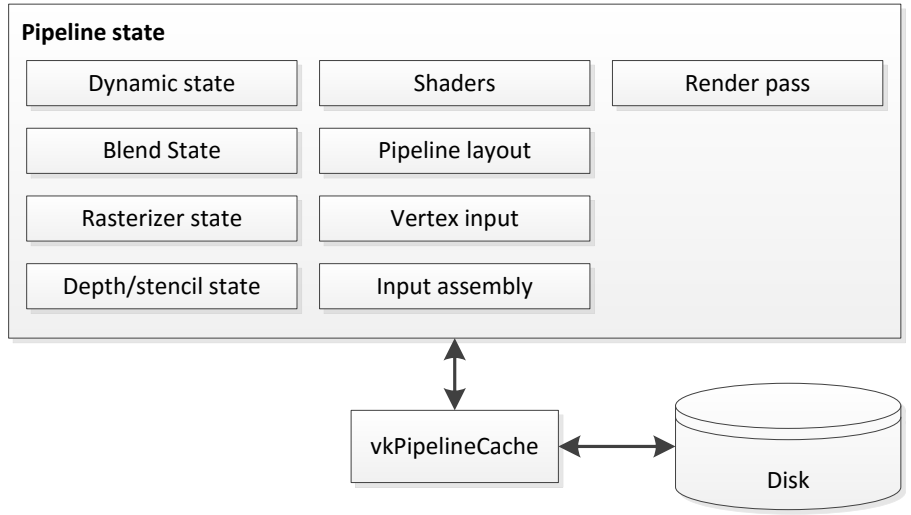
- **In an ideal world…**
  - All pipeline combinations should be created upfront

- **…but this requires detailed knowledge of every potential shader/state combination that you might have in your scene**
  - As an example, one of our fragment shaders have ~9000 combinations
  - Every one of these shaders can use different render state
  - We also have to make sure the pipelines are bound to compatible render passes
  - **An explosion of combinations!**

Pipeline cache

- **Vulkan has built-in support for pipeline caching**
  - Store to disk and re-use on next run

- **Can also speed up pipeline creation during run-time**
  - If the pipeline state is already in the cache it can be re-used

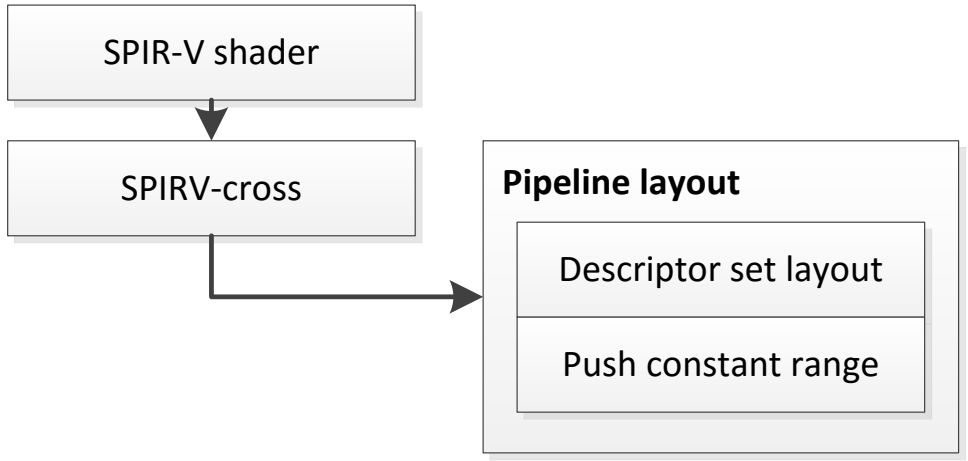


Pipeline layout

- **Defines what kind of resources are in each binding slot in your shaders**
  - Textures, samplers, buffers, push constants, etc
- **Can be shared among different pipeline objects**

Pipeline layout

- Use SPIRV-Cross to automatically get binding information from SPIR-V shaders



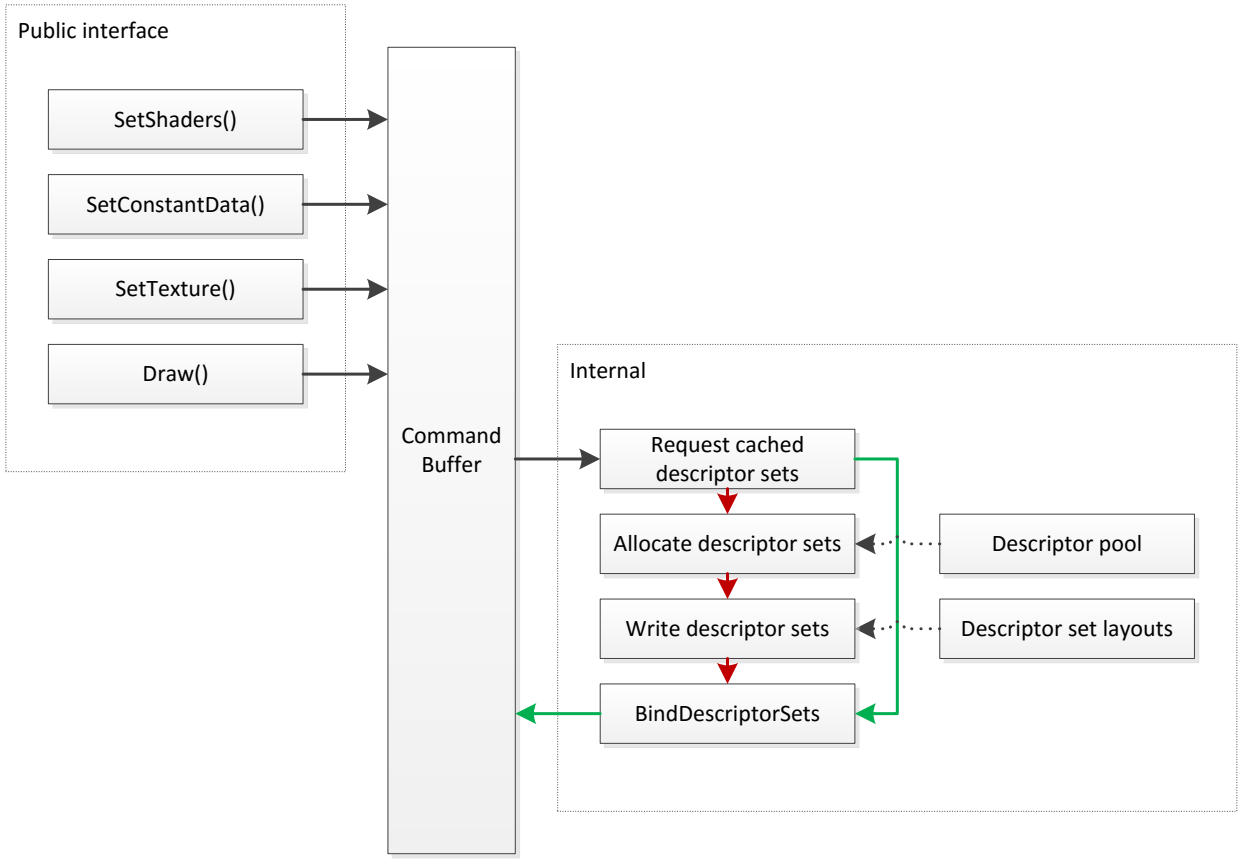
Descriptor Sets

- **Textures, uniform buffers, etc. are bound to shaders in descriptor sets**
  - Hierarchical invalidation
  - Order descriptor sets by update frequency

- **Ideally all descriptors are pre-baked during level load**
  - Keep track of low level descriptor sets per material
  - But, this is not trivial

Descriptor Sets

- **Our solution:**
  - Keep track of bindings and update descriptor sets when necessary
  - Keep cache of descriptor sets used with immutable Vulkan objects

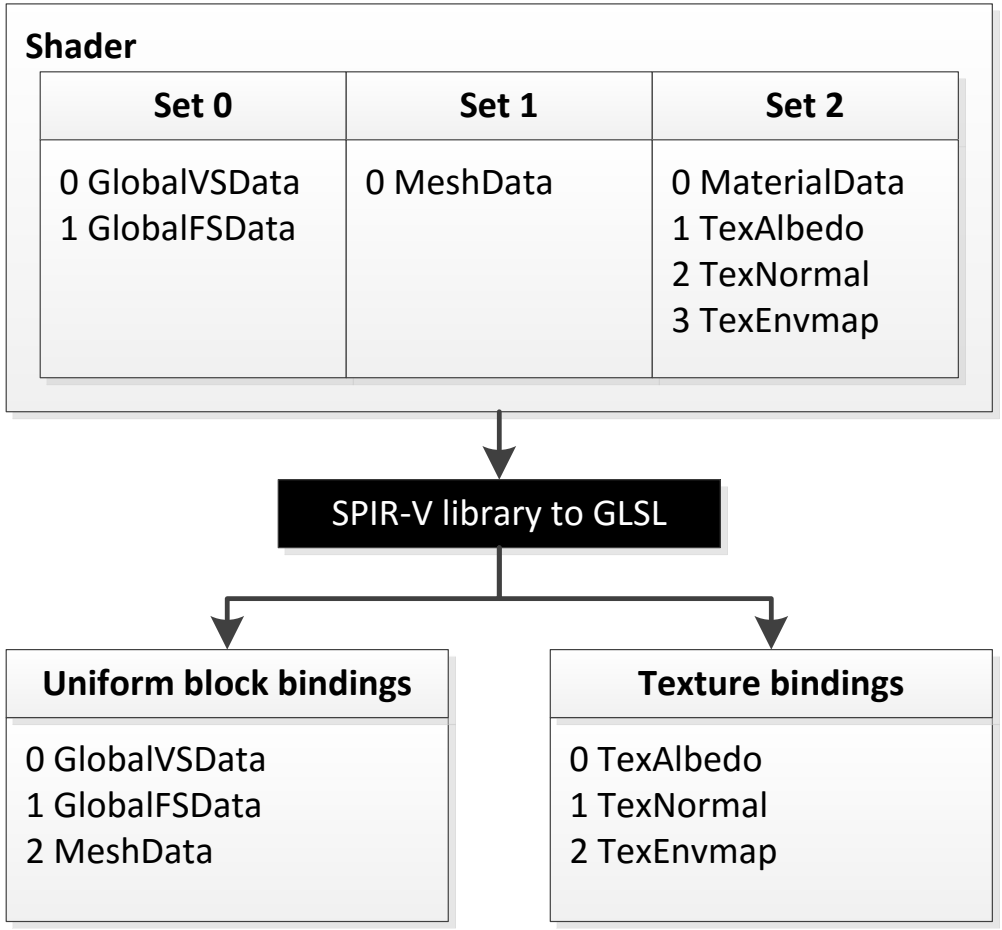


Descriptor Set emulation

- **We also need to support this in OpenGL**

- **Our solution:**
  - Emulate descriptor sets in our OpenGL backend
  - SPIRV-Cross collapses and serializes bindings

Descriptor Set emulation



Push Constants

- **Push constants replace non-opaque uniforms**
  - Think of them as small, fast-access uniform buffer memory
- **Update in Vulkan with vkCmdPushConstants**
- **Directly mapped to registers on Mali GPUs**

Push Constant Emulation

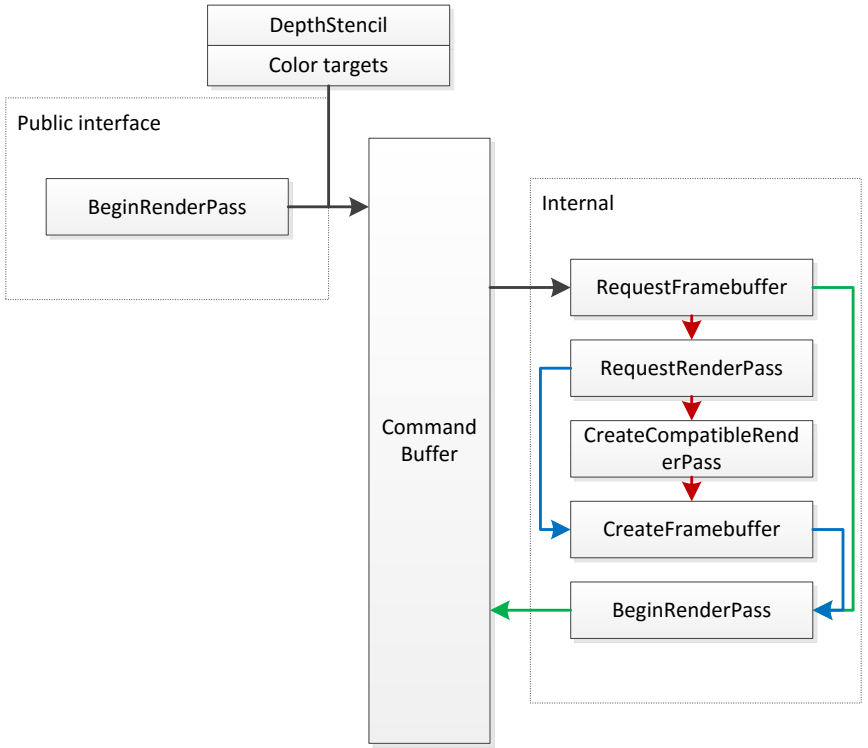
- **But again, we need to support OpenGL as well**

- **Our solution:**
  - Use SPIRV-Cross to turn push constants into regular non-opaque uniforms
  - Logic in our OpenGL/Vulkan backends redirect the push constant data appropriately

Render pass

- **Used to denote beginning and end of rendering to a framebuffer**

- **Can be re-used but must be compatible**
  - Attachments: Framebuffer format, image layout, MSAA?
  - Subpasses
  - Attachment load/store



Subpass Inputs

- Vulkan supports subpasses within render passes
- Standardized GL_EXT_shader_pixel_local_storage!
- Also useful for desktop GPUs

Subpass Input Emulation

- **Supporting subpasses in GL is not trivial, and probably not feasible on a lot of implementations**

- **Our solution:**
  - Use SPIRV-Cross to rewrite subpass inputs to Pixel Local Storage variables or texture lookups
  - This will only support a subset of the Vulkan subpass features, but good enough for our current use

Synchronization

- **Submitted work is completed out of order by the GPU**
- **Dependencies must be tracked by the application and handled explicitly**
  - Using output from a previous render pass
  - Using output from a compute shader
  - Etc
- **Synchronization primitives in Vulkan**
  - Pipeline barriers and events
  - Fences
  - Semaphores

Render passes and pipeline barriers

- **Most of the time the application knows upfront how the output of a renderpass is going to be used afterwards**
- **Internally we have a couple of usage flags that we assign to a render pass**
  - On EndRenderPass we implicitly trigger a pipeline barrier

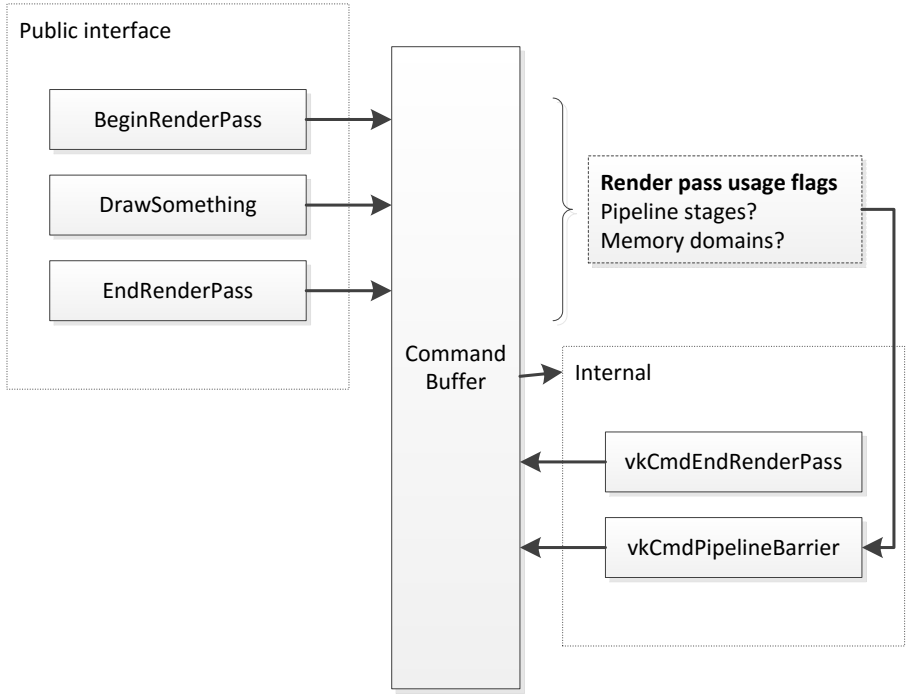


Image Layout Transitions

- **Must match how the image is used at any time**
- **Pedantic or relaxed**
  - Some implementations will require careful tracking of previous and new layout to achieve optimal performance
  - For Mali we can be quite relaxed with this – most of the time we can keep the image layout as VK_IMAGE_LAYOUT_GENERAL

Summary

- Don’t allocate or release during runtime
- Batching still applies
- Multi-thread your code!
- Use push-constants as much as possible
- Multi-pass is fantastic on mobile GPUs

Panel Session - Moving to Vulkan: Lessons to note when going explicit

Tom Olson, ARM
Michael Worcester, Imagination Technologies
Marco Trivellato, Unity Technologies
Jon Kennedy, Intel
Alon Or-bach, Samsung (Chair)

Beer Festival!

Thank you for coming – keep in touch and follow @KhronosUK on Twitter