Vulkan's Key Features on ARM Architecture

ARM

Daniele Di Donato, ARM

GDC 2016

Outline

- Vulkan main features
  - Mapping Vulkan Key features to ARM CPUs
  - Mapping Vulkan Key features to ARM Mali GPUs

Vulkan

- Good match for mobile and tiling architectures
  - Explicit multi-pass render passes
  - No hidden costs (copies, allocs, shader recompiles, etc)
  - Multi-threaded
  - Low overhead

- The driver is lightweight and doesn't execute any error checking or validation
  - No more safety net as in OpenGL….
  - ….but freedom to squeeze performance

Vulkan Main Features for Mobile

- Multi-threading (even mid-range phones now have 4 cores)
  - Most of the function doesn't need to be externally synchronized
  - See chapter 2.4 "Threading Behaviour" of the spec
- Multi-pass Render Passes
  - Able to exploit faster Tile cache memory on mobile
  - See chapter 7 "Render Pass" of the spec
- Other features
  - Independent samplers and textures
    - Ability to use the same Sampler configuration to access multiple textures
  - Low level memory bindings
    - Resource creation doesn't allocate backing memory
  - Sparse memory bindings
    - Backing memory can be assigned completely or partially at runtime (use case: virtual textures)

Multi-Threading in Vulkan

- The Vulkan spec guarantees that some of its core functions don't need to be explicitly synchronized by the programmer (see chapter 2.4 of the Vulkan spec)

- This allows multiple threads to call the same functions or set of functions at the same time

- The typical use-cases are:
  - Command buffer construction: Multiple threads build various command buffers at the same time based on the grouping made by the engine
  - Shaders compilation: Multiple threads compile the shaders used
  - Memory bindings: Multiple threads compute the memory requirements for the textures and allocate/assign it at runtime (multiple virtual-textures update)

Multiprocessing Support in ARM CPUs

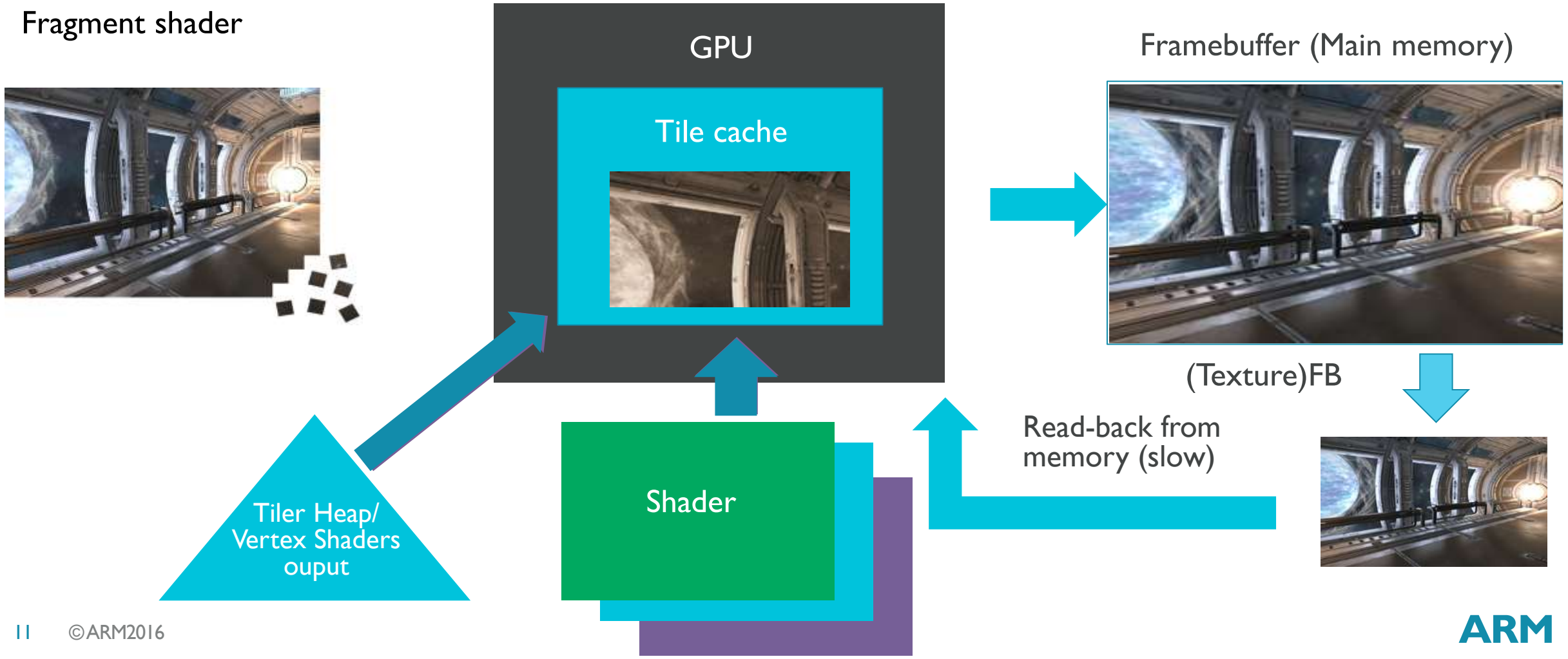
- Inside the single core
  - ARM SIMD Neon
    - Allows vectorization of operations
    - Typically used to speed-up the vector math used for physics, animations, etc.
    - Really useful if the task to solve is sequential and cannot be parallelized or multithread overhead is not worth.
- Across all cores
  - ARM big.LITTLE
    - Able to chose between:
      - big cores: High performance core used for hi-load tasks
      - LITTLE cores: High efficiency cores for low-medium load tasks
    - Schedules and migrates tasks according to the load
    - Provides best trade-off within performance and power consumption

Multi-Pass in Vulkan

- Similar to Pixel Local Storage introduced by ARM
- Especially on Tiled GPUs: allows the driver to perform various optimizations when each pixel rendered in a subpass accesses the results of the previous subpass at the same pixel location
- All the data can be contained and remain on the fast on-chip memory
- Some use-cases:
  - Deferred Rendering
  - Tone-mapping
  - Soft-particles (1st subpass renders the solid geometry and the 2nd renders the particles accessing the depth information)

Deferred Tile-Based Rendering 101

Typical tile-based rendering



Deferred Tile-Based Rendering 101

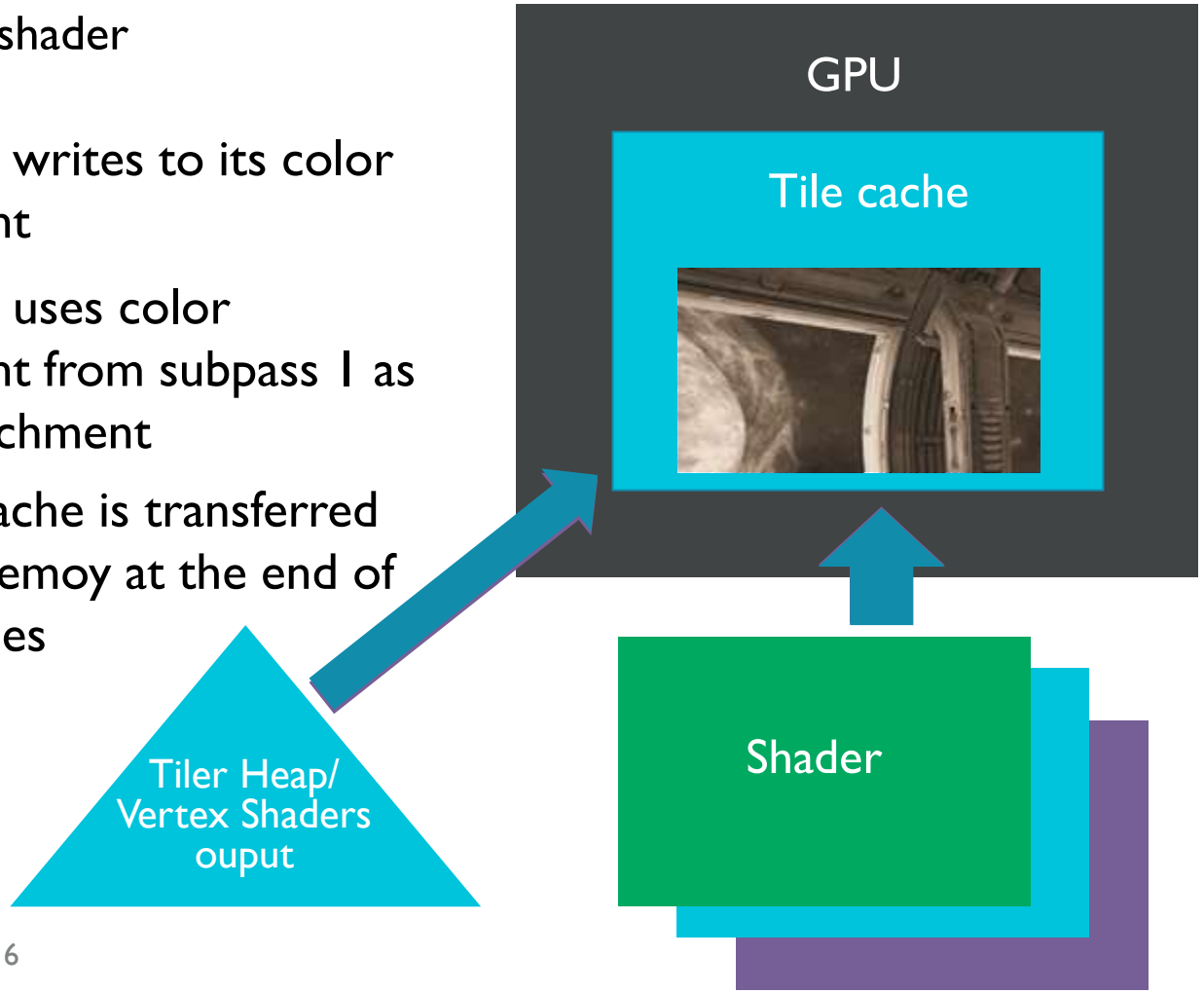
Multi-pass tile-based Rendering

Fragment shader

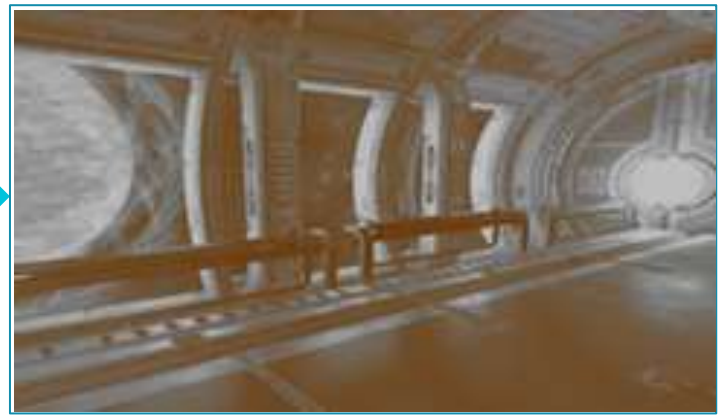
Subpass 1 writes to its color attachment

Subpass 2 uses color attachment from subpass 1 as input attachment

The Tile cache is transferred to main memoy at the end of all subpasses



Framebuffer (Main memory)



Other Mali Features Available

Enabled by you:

- ASTC texture compression
  - Included in the Vulkan core spec
- Early-Z
  - Avoids fragment shading for occluded pixels, sorting front-to-back of opaque geometry gives best results
- 4x MSAA
  - Multisampling algorithm happening on Tile memory for free

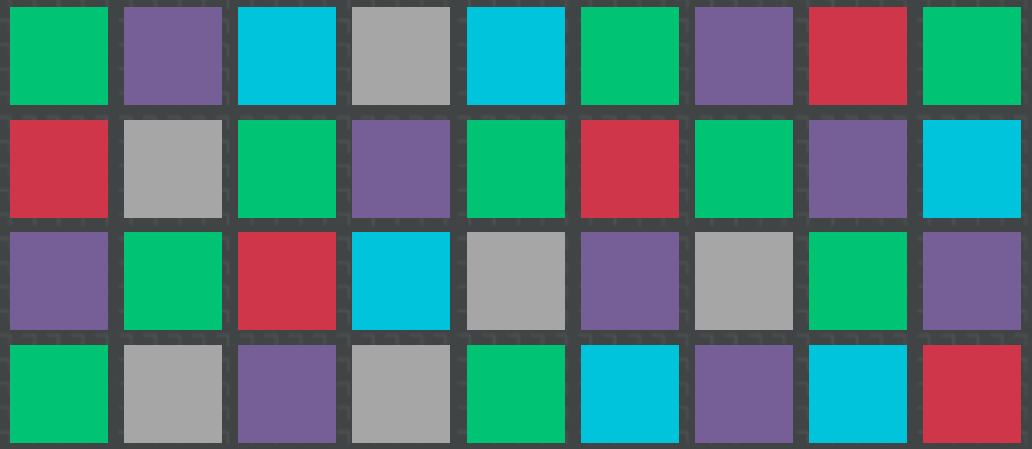
Automatically enabled:

- AFBC (Arm Frame Buffer Compression)
  - Transparently reduces the memory bandwidth required to save energy
- Transaction Elimination
  - Avoids the computations related to a tile if it's unchanged from the previous frame (UI and 2D games with static props will benefit from this feature)
- Forward Pixel Kill

Thank you!

ARM

To Find Out More….

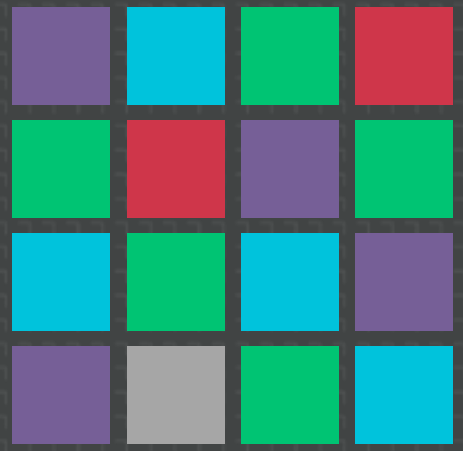


ARM Booth #1624 on Expo Floor:

- Live demos of the techniques shown in this session
- In-depth Q&A with ARM engineers
- More tech talks at the ARM Lecture Theatre

http://malideveloper.arm.com/gdc2016:

- Revisit this talk in PDF and video format post GDC
- Download the tools and resources



More Talks From ARM at GDC 2016

Available post-show at the Mali Developer Center: malideveloper.arm.com/

ARM EPIC GAMES SAMSUNG

Vulkan on Mobile with Unreal Engine 4 Case Study

Weds. 9:30am, West Hall 3022

ARM NINJA enlighten

Making Light Work of Dynamic Large Worlds

Weds. 2pm, West Hall 2000

ARM unity nDreams

Achieving High Quality Mobile VR Games

Thurs. 10am, West Hall 3022

ARM

Optimize Your Mobile Games With Practical Case Studies

Thurs. 11:30am, West Hall 2404

ARM enlighten allegorithmic

An End-to-End Approach to Physically Based Rendering

Fri. 10am, West Hall 2020

ARM

Marius Bjørge
Graphics Research Engineer

GDC 2016

Agenda

- Overview
- Command Buffers
- Synchronization
- Memory
- Shaders and Pipelines
- Descriptor sets
- Render passes

Overview – OpenGL

- OpenGL is mainly single-threaded
  - Drawcalls are normally only submitted on main thread
  - Multiple threads with shared GL contexts mainly used for texture streaming
- OpenGL has a lot of implicit behaviour
  - Dependency tracking of resources
  - Compiling shader combinations based on render state
  - Splitting up workloads
  - All this adds API overhead!

Overview – Vulkan

- Vulkan is designed from the ground up to allow efficient multi-threading behaviour
- Vulkan is explicit in nature
  - Applications must track resource dependencies to avoid deleting anything that might still be used by the GPU or CPU
  - Little API overhead
- Vulkan is very verbose in terms of lines of code
  - Getting a simple "Hello Triangle" running requires ~1000 lines of code

Overview – Vulkan

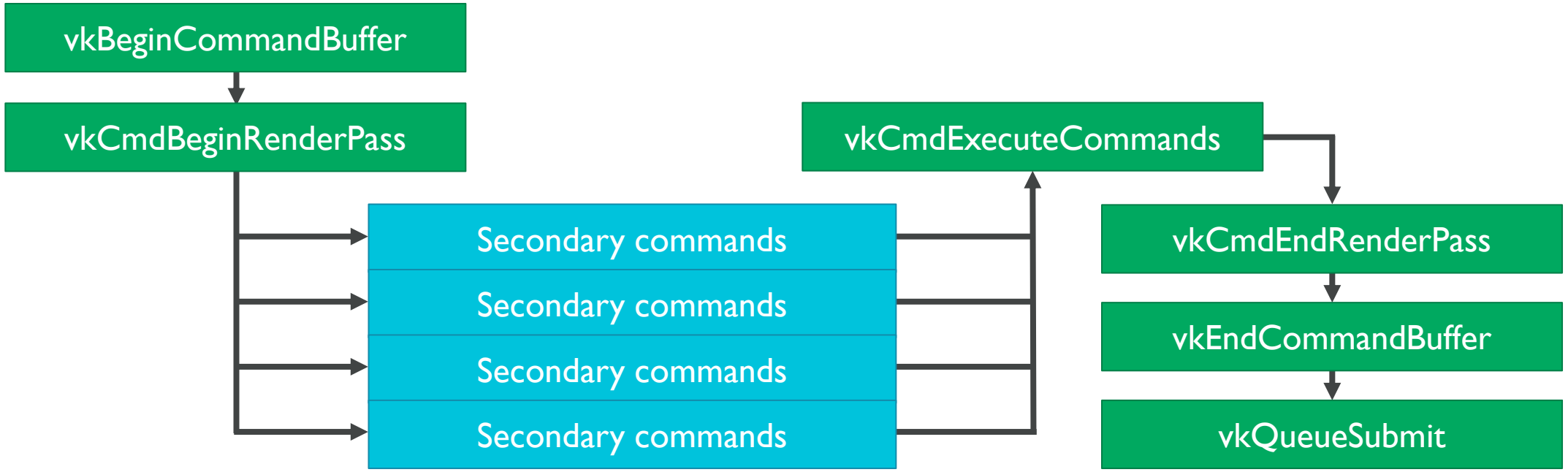
- To get the most out of Vulkan you probably have to think about re-designing your graphics engine
- Migrating from OpenGL to Vulkan is not trivial
- Some things to keep in mind:
  - Do you really need Vulkan for your project?
  - Portability?

Command Buffers

- Used to record commands which are later submitted to a device for execution
  - This includes draw/dispatch, texture uploads, etc.
- Primary and secondary command buffers

- Command buffers work independently from each other
  - Contains all state
  - No inheritance of state between command buffers

Command Buffers



Synchronization

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

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

- 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

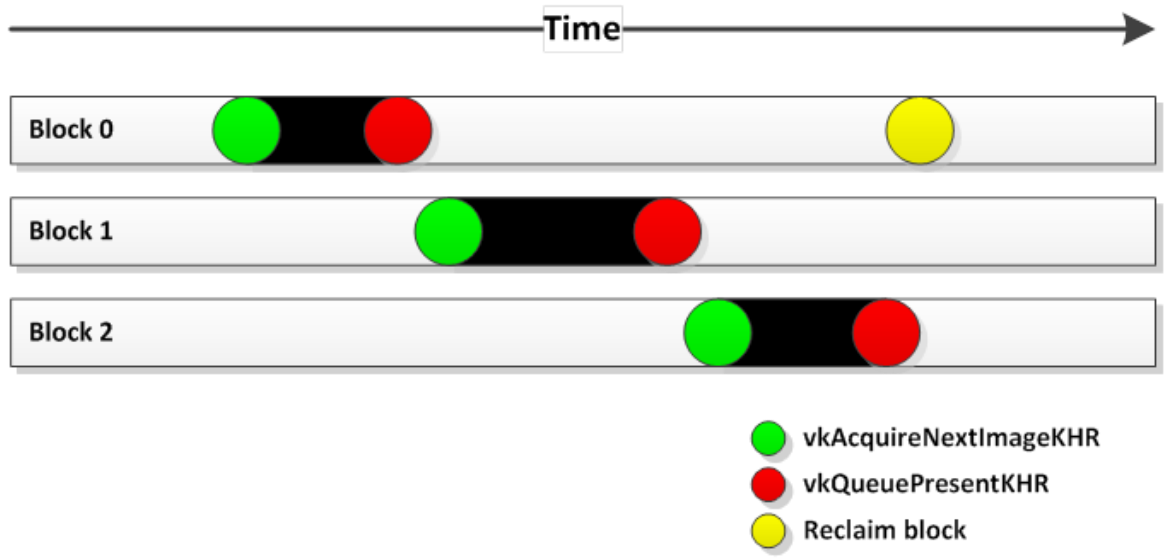


Image Layout Transitions

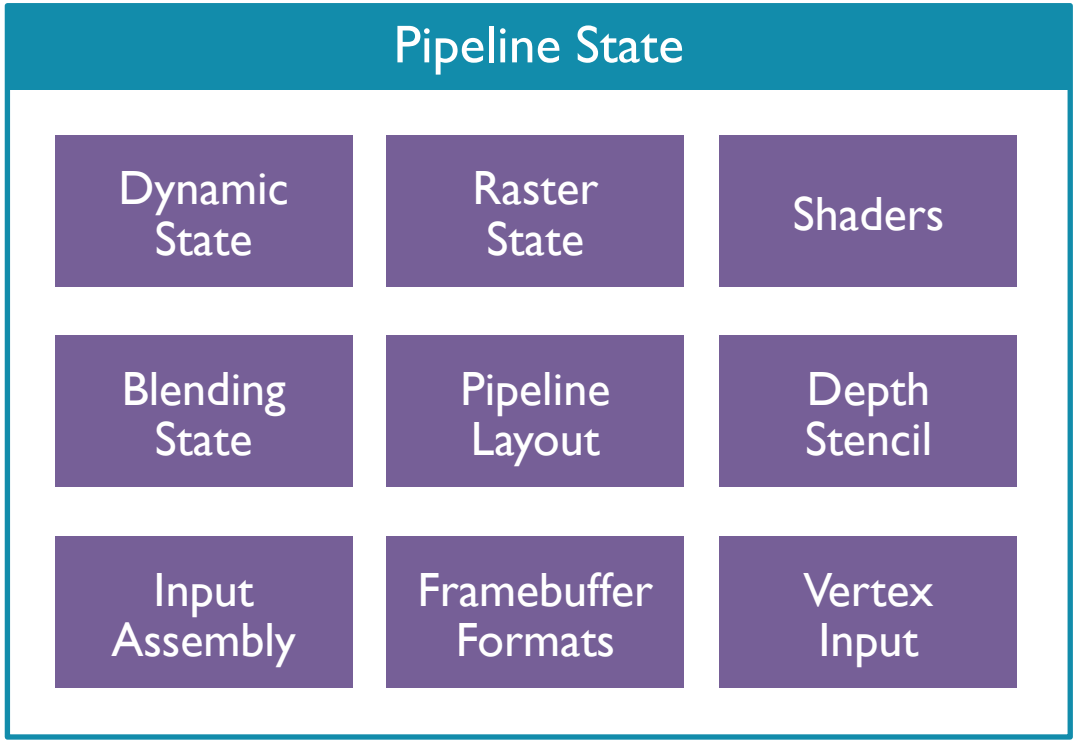
- Must match how the image is used at any time
- Pedantic or relaxed
  - Some implementations might require careful tracking of previous and new layout to achieve optimal performance
  - For Mali we can be quite relaxed with this – most of the time we can keep the image layout as VK_IMAGE_LAYOUT_GENERAL

Pipelines

- Vulkan bundles state into big monolithic pipeline state objects
- Driver has full knowledge during shader compilation

```
vkCreateGraphicsPipelines(...)
;

vkBeginRenderPass(...);
vkCmdBindPipeline(pipeline);
vkCmdDraw(...);
vkEndRenderPass(...);
```



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, a typical fragment shader in a graphics engine such as Unreal may have ~9 000 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

- Result of the pipeline construction can be re-used between pipelines
- Can be stored out to disk and re-used next time you run the application

Shaders

- Vulkan standardized on SPIR-V
- Khronos reference compiler
  - Outputs SPIR-V from your GLSL shader sources
  - GL_KHR_vulkan_glsl
  - Can be easily integrated into your graphics engine

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
- Simple solution:
  - Keep track of bindings and update descriptor sets when necessary
  - Keep around cache for non-dynamic descriptor sets

SPIR-V reflection

- Introducing SPIR2CROSS
  - Convert SPIR-V to readable GLSL
  - https://github.com/ARM-software/spir2cross

- Using SPIR2CROSS we can retrieve information about bindings as well as inputs and outputs directly form the SPIR-V binary
  - This is useful information when creating or re-using existing pipeline layouts and descriptor set layouts
  - Also allows us to easily re-use compatible pipeline layouts across a bunch of different shader combinations

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

```
// 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;
```

Render Passes

- Describes the beginning and end of rendering to a framebuffer
- Render passes in Vulkan are very explicit
  - Declare when a render pass begins
    - Load, discard or clear the framebuffer?
  - Declare when a render pass ends
    - Which parts do you need to be committed to memory?

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;
...
```

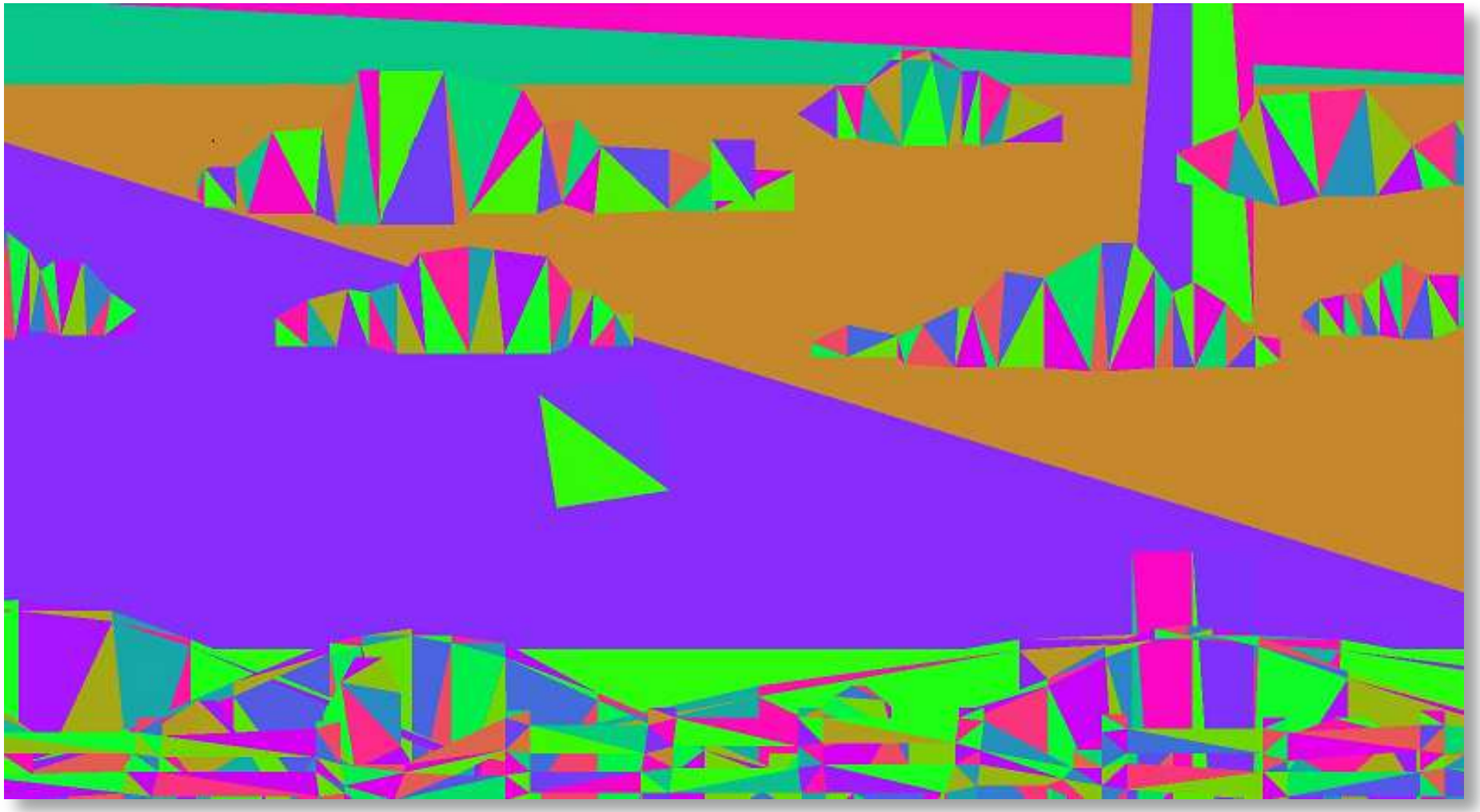
Niklas “Smedis” Smedberg
Technical Director, Platform Partnerships

GDC 2016

UE4 ProtoStar Demo

- **Goals:**
  - Impressive real-time graphics to showcase UE4 Vulkan on Samsung Galaxy S7
  - Must be the best in the world
- **Problems to overcome:**
  - Mobile graphics features did not exist in UE4 yet
  - Vulkan API did not exist yet
  - Driver did not exist yet
  - Device did not exist yet
  - Not enough time or people

UE4 Vulkan: A First Attempt



UE4 Vulkan: Final Results

UE4 Vulkan Thoughts

- **One queue**
  - One big command buffer per frame (array round-robin reused)
  - Multi-threaded rendering a great future opportunity in Vulkan
- **“External synchronization”**
  - Semaphores for synchronization on GPU (GPU wait / ordering)
  - Fences for synchronization on CPU (CPU wait / check for completion)
- **Recording command buffer**
  - Simplest usage-case: Instancing
  - Nice usage-case: VR (stereoscopic rendering)

UE4 Vulkan Source Code

- **UE4 Vulkan source code on github soon!**
  - unrealengine.com
- **Read and learn**
  - Experiment with Vulkan
  - Make something fun!

Galaxy

Jungwoo Kim
Principle Engineer, Samsung Mobile Graphics Team

GDC 2016

- 3 year long-term project started in 2012 by Samsung
- 1.5 year contribution for Vulkan within Khronos group
- 1 year collaboration with our partners for the demo
- But this is just the beginning…

- Samsung Mobile is planning a game developer support program
- Official announcement at Samsung Developer Conference in April
- Samsung wants to engage with the game developer community
- Samsung also wishes to support Vulkan game developers

Galaxy Vulkan™

GameDev

#GalaxyGameDev