VULKAN FAST PATHS

GDC 2016

AMD

CONTENTS

- Binding Model
- Render Passes
- Barriers & Sync

VULKAN FAST PATHS – BINDING MODEL

TIMOTHY LOTTES (@TimothyLottes)

AMD

VULKAN FAST PATHS: BINDING MODEL

BY TIMOTHY LOTTES - AMD DEVTECH

- Speaker's background
  - Involved in Vulkan binding model design as a Khronos Member when working at Epic Games
    - Quite a challenge to design a model which works well on all GPU hardware from mobile to desktop
  - Now an advocate of Vulkan working in AMD DevTech

- Goal of the Binding Model Part of this talk
  - Help you get great performance in Vulkan on AMD hardware with a simple design
    - Provide low-level background on the GCN hardware binding model
    - Present how the Vulkan binding model maps to AMD hardware
    - Kickstart thinking about the flexibility of Vulkan outside the confines of prior graphics APIs
    - Ultimately to provide an intuition on how to design for performance

- Going quite low-level
    - Feel free after the talk to contact and ask questions: Timothy.Lottes@amd.com

GCN TERMS

BACKGROUND ON THE HARDWARE

- SGPR – 32-bit Scalar General Purpose Register
- VGPR – 32-bit Vector General Purpose Register
- K$ – Scalar Data Cache
- CU – Compute Unit
  - Each CU as a throughput of 64 {d=a*b+c} operations per clock
- Wave – 64 shader invocations running in lock-step

- The following instructions can multi-issue at the same time on different waves in a CU,
  - SMEM – Scalar Memory Instruction (access to the K$, buffer access via dynamically uniform addresses)
  - SALU – Scalar Arithmetic Instruction
  - VMEM – Vector Memory Instruction (image and buffer access)
  - VALU – Vector Arithmetic Instruction

DESCRIPTOR TYPES

HOW THEY MAP TO GCN HARDWARE

- Maps to GCN 16-byte Sampler Descriptor
  - VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER (just the Sampler part)
  - VK_DESCRIPTOR_TYPE_SAMPLER
- Maps to GCN 16-byte Buffer Descriptors
  - VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
  - VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
  - VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
  - VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
  - VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
  - VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
- Maps to GCN 32-byte Image Descriptors (later-gen GDC uses 32-bytes for all Image Descriptors)
  - VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER (just the image part)
  - VK_DESCRIPTOR_TYPE_IMAGE_ATTACHMENT
  - VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
  - VK_DESCRIPTOR_TYPE_STORAGE_IMAGE

USER-DATA SGPRS

GDC HARDWARE BACKGROUND FOR VULKAN DESCRIPTOR MODEL

- Shaders have access to 2 register sets (GPR = General Purpose Register)
  - Scalar or SGPRs which are dynamically uniform (ie the same) for all invocations in a wave
    - These are used for example for Constants and Descriptors
  - Vector or VGPRs which are unique for all invocations in a wave

- GCN has ability to pre-load up to 16 SGPRs prior to wave launch
  - These are known as USER-DATA SGPRs
  - There is some overhead in setting USER-DATA in both GPU command buffer, and right before wave launch
    - So don’t use more than required
  - A few of these USER-DATA SGPRs are used internally by the driver
    - Varies by shader stage and features used, and might change in future drivers and/or hardware
  - The rest are used for the Vulkan binding model
    - Push Constants
    - 32-bit Descriptor Set pointers (Sets are kept in lower 32-bit address space)
    - UNIFORM_BUFFER_DYNAMIC or STORAGE_BUFFER_DYNAMIC Descriptors (each take 4 USER_DATA SGPRs)

USER-DATA SGPRS: AND SPILLING

GDC HARDWARE BACKGROUND FOR VULKAN DESCRIPTOR MODEL

- After USER-DATA SGPRs fill up, the driver fills a driver managed buffer with the overflow
  - Spilling does have an associated increase in cost (indirection in the shader, etc)
- USER-DATA SGPR fill priority is driver dependent (may change based on optimizations, driver revision)
  - Current priorities are generally:
    - {1st Push Constants, then Sets {Dynamic Descriptors, followed by Set pointer} from Set 0 to N}
- Example of possible USER-DATA SGPR usage,
  - 4 for internal driver usage
  - 2 for two 32-bit Push Constants
  - 4 for one Dynamic Descriptor for Set=0
  - 1 for one 32-bit Set pointer for Set=0
  - 1 for one 32-bit Set pointer for Set=1

- General takeaway,
  - Try to keep the number of {Push Constants, bound Sets and Dynamic Descriptors} low enough to not spill

CONSTANT AND DESCRIPTOR LOADS ON GCN

HARDWARE BACKGROUND

- Constants and Descriptors are block loaded by SMEM operations
  - This also can apply to any Buffer loads which are known to be from dynamically uniform addresses
  - Supported address modes: base from descriptor + either register or immediate offset
    - S_BUFFER_LOAD_DWORD* destination, descriptor, SGPR_provided_offset
    - S_BUFFER_LOAD_DWORD* destination, descriptor, immediate_20bit_offset
      - 20-bits = 1 MB, so for large Buffers or Sets, keep immediate offset accessed data at the beginning and dynamic accessed data afterwards
- S_BUFFER_LOAD_DWORD* can block load {1,2,4,8 or 16} 32-bit values in one instruction
  - Driver can coalesce multiple Constant loads into one larger block load
  - Best to keep Constants grouped by locality of usage and block aligned to support this
  - Blocks loads make dynamic base addresses low cost on GCN,
    - // Example, uses one extra SALU operation for a dynamic base for 8 32-bit constants
    - S_ADD_U32 offset, base + immediate_offset
    - S_BUFFER_LOAD_DWORDX8 destination, descriptor, offset
- Descriptors are loaded via S_LOAD_DWORD* (uses pointer instead of Descriptor for base)
  - Same address modes and block load support as S_BUFFER_LOAD_DWORD*

ONE SET DESIGN

FAST PATH, “BINDLESS” OR RATHER “BIND-EVERYTHING” ON VULKAN

- Place all Descriptors in one giant Descriptor Set
  - layout (set=0, binding=N) uniform texture2D textures[hugeNumber]
- Leave the one giant Descriptor Set always bound
  - No more vkCmdBindDescriptorSets() calls for each draw/dispatch
  - Instead use Push Constant(s) via vkCmdPushConstants() for per-draw indexes into binding array
- Per-draw frequency base index via Push Constant: textures[pushConstant+1]
  - S_ADD_U32 arrayBase, setBase, immediate_32bit_offset
    - This instruction is effectively free, and also not required for the second texture from the array binding
  - S_LOAD_DWORDX8 textureDescriptor, arrayBase, immediate_20bit_offset
- Per-frame frequency Textures can be immediate indexed: textures[2]
  - S_LOAD_DWORDX8 textureDescriptor, setBase, immediate_20bit_offset
  - Best to keep perf-frame frequency textures towards the base of the set (only 20-bit immediate)

DYNAMIC DESCRIPTORS

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC & VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC

- Dynamic base address(es) provided in
  - vkCmdBindDescriptorSets(… dynamicOffsetCount, pDynamicOffsets)

- Driver builds a unique GCN Buffer Descriptor based on the dynamic offset for the bind call
  - This 16-byte Buffer Descriptor is placed in USER-DATA SGPRs if possible

- Takeaway
  - For per-draw frequency, this has a good amount of overhead (CPU work, plus space in USER-DATA)
    - Try Push Constants instead (next slide)
  - However Dynamic Descriptors get USER-DATA placement
    - So we can use this to remove an indirection in the shader
    - And get to all Constants without having to load a Descriptor first

ONE DYNAMIC BUFFER DESCRIPTOR DESIGN

FAST PATH, “BIND-EVERYTHING” APPLIED TO CONSTANT DATA

- Example case: draws which need to source {per-frame, per-pass, and per-draw} Constants

- Possible to optimize this to
  - One VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC which stays the same each draw
    - Extension of “One Set Design”, Dynamic Descriptor built when Set is bound (per-pass frequency instead of per-draw)
    - Removes the indirection, Dynamic Descriptor in USER-DATA
  - One 32-bit Push Constant which changes per-draw and supplies the per-draw offset

- Each pass (few passes per frame) gets a separate UNIFORM_BUFFER_DYNAMIC Descriptor
  - Buffer contents: [per-frame] [per-pass] [draw0] [draw1] [draw2] . . . [drawN]
  - Per-frame data is duplicated for each pass and can be accessed with immediate offsets
  - Per-pass data can be accessed with immediate offsets
  - Per-draw uses the dynamic base offset supplied in the Push Constant
    - Fast, GCN block loads Constants

IMAGE DESCRIPTORS: SAMPLED VS STORAGE

OPTIMAL DESCRIPTOR CHOICE

- The read-only SAMPLED_IMAGE can be faster than using STORAGE_IMAGE just for reading
  - Even though these share the same descriptor type in GCN hardware
  - Still important to use correct Vulkan descriptor type

  - VK_DESCRIPTOR_TYPE_STORAGE_IMAGE needs to be in VK_IMAGE_LAYOUT_GENERAL
    - Not compressed

  - VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE can be in VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
    - Can support compression
    - GCN3 (Tonga/Antigua/Fiji) added Delta Color Compression (DCC) for render targets

  - DCC can also be used for standard read-only non-block-compressed formatted textures (RGBA8, RGBA16F, etc)
    - Key is to adjust texture upload process
    - Use vkCmdCopyBufferToImage() in a graphics queue with VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
    - Uses pixel shader internally to copy from buffer to image, gets DCC compressed on output

IMMUTABLE SAMPLERS

NO-LOAD SAMPLERS

- Ability to specify immutable Samplers in Descriptor Set Layout
  - VkDescriptorSetLayoutBinding.pImmutableSamplers
- Provides Sampler data at PSO creation time
  - Sampler can be compiled into the Shader

- Immutable Samplers can be constructed by SALU instructions instead of SMEM loads
  - Reduces the amount of latency in the shader
  - SALU pipe is mostly under-utilized

DESCRIPTOR SET LAYOUT

HOW DESCRIPTORS ARE FILLED IN DESCRIPTOR POOL GPU MEMORY

- Descriptors are read through the scalar data cache (K$) which has 64-byte cache lines
  - Can get 2 Image Descriptors per cache line, 4 Buffer Descriptors per cache line, or a mix
  - For large Sets best to not do sparse random access of Descriptors, instead group to maintain locality of usage
  - For “One Set Design”, best to sub-allocate to keep cache line aligned locality of usage
- Descriptors are packed into GPU memory in order of how they appear in VkDescriptorSetLayoutBinding
  - Example layout to GPU memory mapping

Set Layout	Descriptor(s) Bytes	Memory Offset In Set	K$ Cacheline
layout(set=0,binding=0) uniform sampler s0;	16	0	0
layout(set=0,binding=1) uniform samplerBuffer sb0;	16	16	0
layout(set=0,binding=2) uniform texture2D t0;	32	32	0
layout(set=0,binding=3) uniform samplerBuffer sb1[4];	16*4=64	64	1
layout(set=0,binding=4) uniform texture2D t1[2];	32*2=64	96	2

DESCRIPTOR POOLS

USED TO ALLOCATE DESCRIPTOR SETS

- VkDescriptorPoolCreateInfo.flags = 0
  - Not using VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT
    - Pools which support vkFreeDescriptorSets() go down a driver managed path which can have fragmentation
    - Uses driver managed dynamic memory allocator
  - Suggest using the path that only supports vkResetDescriptorPool()
    - Descriptor Pool becomes a pre-allocated chunk of GPU and CPU memory
    - Allocation is like increasing an offset
  - Be mindful of setting reasonable limits
    - VkDescriptorPoolCreateInfo.maxSets effects amount of CPU memory
  - Descriptor Pools on Windows have variable mapping based on resource utilization
    - Keeping reasonable limits enables the fastest path
    - Fastest path shares the 256 MB maximum window of directly accessible GPU memory
      - GCN descriptors are typically at most 32-bytes (ballpark 32K descriptors per MB of GPU memory)
      - vkUpdateDescriptorSets() writes directly into the Descriptor Pool GPU memory
      - Using VkCopyDescriptorSet* can result in the CPU reading GPU memory then writing GPU memory (not as fast as just writing)

UPDATING DESCRIPTOR SETS

GETTING PIPELINED UPDATE, KEY FOR TEXTURE STREAMING WITH “ONE SET DESIGN”

- vkUpdateDescriptorSets() effects can be immediate (by function return, GPU memory writes in progress)
  - So cannot write Descriptors which might be in use
  - Challenge for texture streamers
    - Would like to update a Descriptor between frames
- Workaround for pipelined update, example for minimum latency VR
  - Keep 2 copies of the Descriptor Set
    - Switch between each every other frame
    - Add necessary synchronization to avoid updating when in use
  - Updates are done to the other copy, so next frame gets the new Descriptor(s)
    - Next frame then does a second update to the other copy, so next-next frame also gets the new Descriptor(s)
  - Can call vkUpdatesDescriptorSets() from multiple threads as long as updates don’t alias same Descriptors

SUMMARY

"BIND-EVERYTHING"

- Showed a fast path which also an simple path in Vulkan
  - Keeps on the no-spill USER-DATA fast path
  - No complex Set management, duplicated Set with simple pipelined update model
  - One Set Layout, no complexities with Layout compatibility
  - Removal of the majority of Set binding calls
  - Avoids the overhead of rebuilding unique Sets each frame for each material
  - Etc

- For continued discussion, questions, comments, and feedback: Timothy.Lottes@amd.com

VULKAN FAST PATHS – RENDER PASSES

GRAHAM SELLERS (@grahamsellers)

AMD

RENDERPASSES

WHAT ON EARTH IS THAT?

- Renderpasses are chunks of back to back GPU work
  - Represented by a Vulkan object
  - Contain one or more sub-passes
  - All rendering happens inside a renderpass
    - Even if it has only a single subpass
  - Dependencies between subpasses are part of the renderpass
    - Driver can schedule work based on future knowledge
    - Driver generates a DAG from dependency information

- Renderpasses are a time machine for drivers!

RENDERPASS IN WORDS

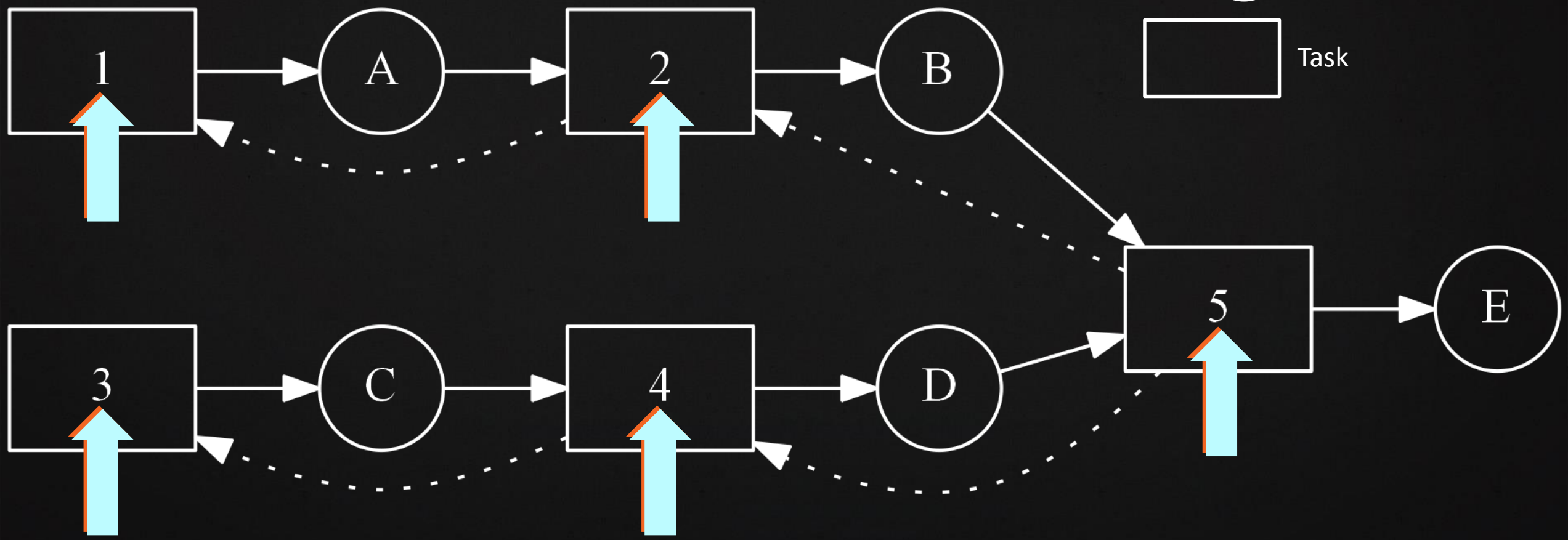
A THOUSAND WORDS

- Consider the following:
  - Subpass 1 produces resource A…
  - Which is consumed by subpass 2, producing resource B
  - Subpass 3 produces resource C…
  - Which is consumed by subpass 4, producing resource D
  - Finally, subpass 5 consumes resources B and D, producing final output E
- Blah, blah, blah; loads of text
  - But this is what API order calls look like

RENDERPASS IN PICTURES

SEEMS LIKE A FAIR TRADE

- Here's the DAG:



CREATING RENDERPASSES

OK. HOW DO I MAKE ONE?

- Simple API – **`vkCreateRenderpass`**

```
VkResult vkCreateRenderPass(
    VkDevice                                    device,
    const VkRenderPassCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkRenderPass*                               pRenderPass);
```

- Creates a renderpass object
  - Usable by **`device`**
  - Using information in **`pCreateInfo`**

RENDERPASS INFORMATION

SO, WHAT'S IN A RENDERPASS?

- Magic is in the **`VkRenderPassCreateInfo`** structure:

```
typedef struct VkRenderPassCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkRenderPassCreateFlags            flags;
    uint32_t                           attachmentCount;
    const VkAttachmentDescription*     pAttachments;
    uint32_t                           subpassCount;
    const VkSubpassDescription*        pSubpasses;
    uint32_t                           dependencyCount;
    const VkSubpassDependency*         pDependencies;
} VkRenderPassCreateInfo;
```

- Arrays of attachments, subpasses and dependency information

ATTACHMENTS

WHERE AM I DRAWING?

- An array of **`VkAttachmentDescription`** structures:

```
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
} VkAttachmentDescription;
```

- Any number of attachments can be used by a renderpass
  - They are referenced by subpasses

ATTACHMENTS

WHERE'S THE DATA?

- Each attachment contains the following:
  - Format and sample count
  - Load operation – where to get the data from (memory, clear, or don't care)
  - Store operation – where to leave the data (memory, or don't care)
    - There are separate load and store operations for stencil
  - Expected layout at the beginning and end of the renderpass
    - Driver will insert layout changes for you

SUBPASSES

WHICH BIT AM I DRAWING?

- An array of **`VkSubpassDescription`** structures:

```
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags       flags;
    VkPipelineBindPoint             pipelineBindPoint;
    uint32_t                        inputAttachmentCount;
    const VkAttachmentReference*    pInputAttachments;
    uint32_t                        colorAttachmentCount;
    const VkAttachmentReference*    pColorAttachments;
    const VkAttachmentReference*    pResolveAttachments;
    const VkAttachmentReference*    pDepthStencilAttachment;
    uint32_t                        preserveAttachmentCount;
    const uint32_t*                 pPreserveAttachments;
} VkSubpassDescription;
```

- References color, depth-stencil, input, and resolve attachments

COLOR ATTACHMENTS

THIS IS WHERE YOUR DATA GOES

- Array of color attachments

```
typedef struct VkSubpassDescription {
    ...
    uint32_t                         colorAttachmentCount;
    const VkAttachmentReference*     pColorAttachments;
    ...
} VkSubpassDescription;
```

- These are your normal color attachments
  - The total number of attachments in the renderpass is unlimited*
  - The number of color attachment references per-subpass is limited

*well, not actually unlimited, but you get the idea

INPUT ATTACHMENTS

READ FROM PREVIOUS SUBPASSES

- Array of input attachments

```
typedef struct VkSubpassDescription {
    ...
    uint32_t                        inputAttachmentCount;
    const VkAttachmentReference*    pInputAttachments;
    ...
} VkSubpassDescription;
```

- Input attachments are the outputs of previous subpasses
  - Represents a data dependency between subpasses

OTHER ATTACHMENTS

MORE STUFF

- Resolve, preserve and depth-stencil attachments:

```
typedef struct VkSubpassDescription {
    ...
    const VkAttachmentReference*    pResolveAttachments;
    const VkAttachmentReference*    pDepthStencilAttachment;
    uint32_t                        preserveAttachmentCount;
    const uint32_t*                 pPreserveAttachments;
} VkSubpassDescription;
```

- Resolve attachments: Where MSAA attachments get resolved to
- DepthStencilAttachment: Depth and stencil
- Preserve attachments: List of attachments that must be preserved

ADDITIONAL DEPENDENCIES

EVEN MORE INFORMATION

- Array of additional dependency information

```
typedef struct VkRenderPassCreateInfo {
    ...
    uint32_t                        dependencyCount;
    const VkSubpassDependency*      pDependencies;
} VkRenderPassCreateInfo;
```

```
typedef struct VkSubpassDependency {
    uint32_t                srcSubpass;
    uint32_t                dstSubpass;
    VkPipelineStageFlags    srcStageMask;
    VkPipelineStageFlags    dstStageMask;
    VkAccessFlags           srcAccessMask;
    VkAccessFlags           dstAccessMask;
    VkDependencyFlags       dependencyFlags;
} VkSubpassDependency;
```

- Used for side effects
  - Stores to images or buffers consumed by later subpasses, for example

GRAPH BUILDING

CAN YOU DAG IT?

- Driver uses renderpass structures to form a DAG
  - Subpasses produce and consume data
  - Resource barriers inserted automatically by driver
  - Scheduling information generated at renderpass creation time
- A DAG of one node isn’t helpful
  - Need renderpasses to include multiple subpasses to be useful

BUT WAIT, THERE'S MORE

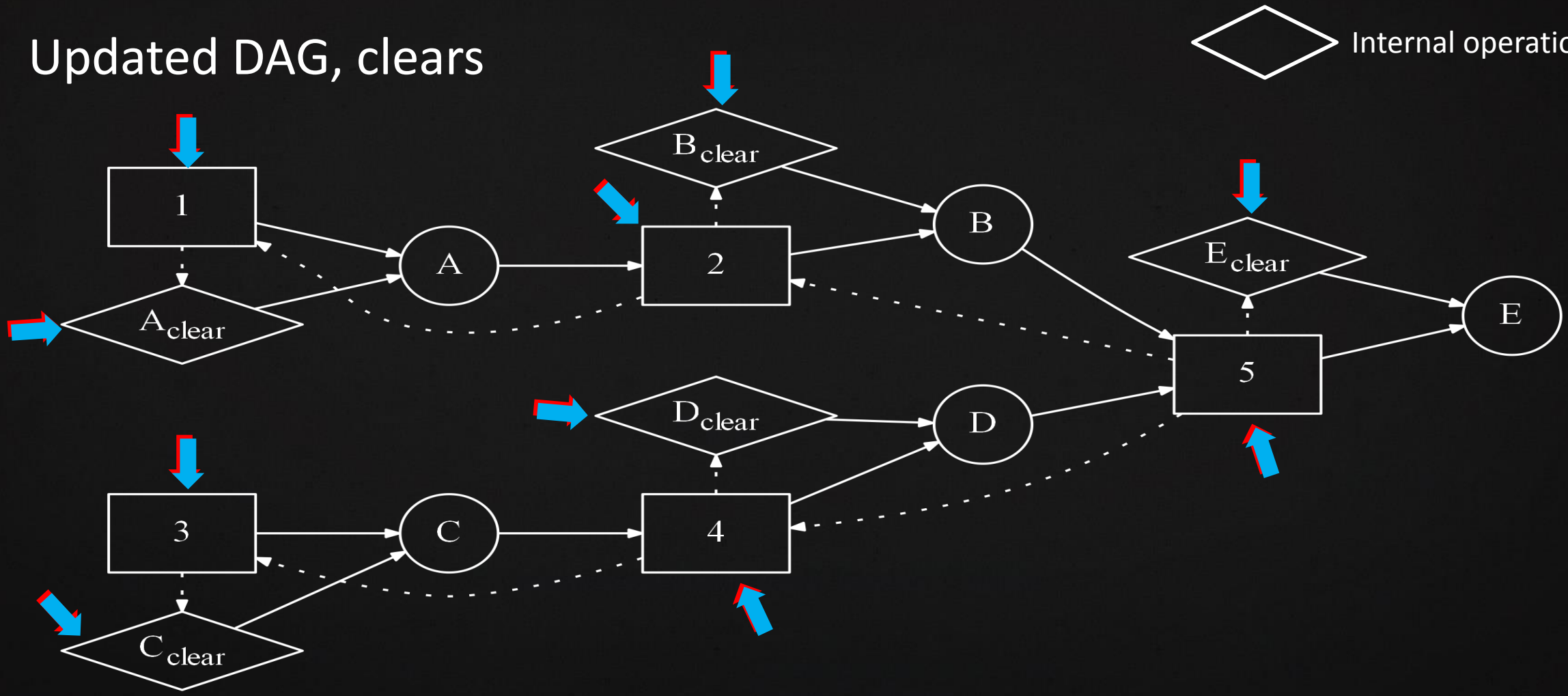
ORDER IN THE NEXT 20 MINUTES

- Internal driver operations
  - Attachments have initial and final states
    - Clears are part of beginning a subpass, for example
  - Attachments go from being outputs to being inputs
    - Flush color caches, invalidate texture caches, change layouts, insert fences
  - Some surfaces require more attention
    - Compressed depth not directly readable by shaders, for example
    - Requires internal driver decompression

LOAD OPS

LET'S MAKE THINGS CLEAR

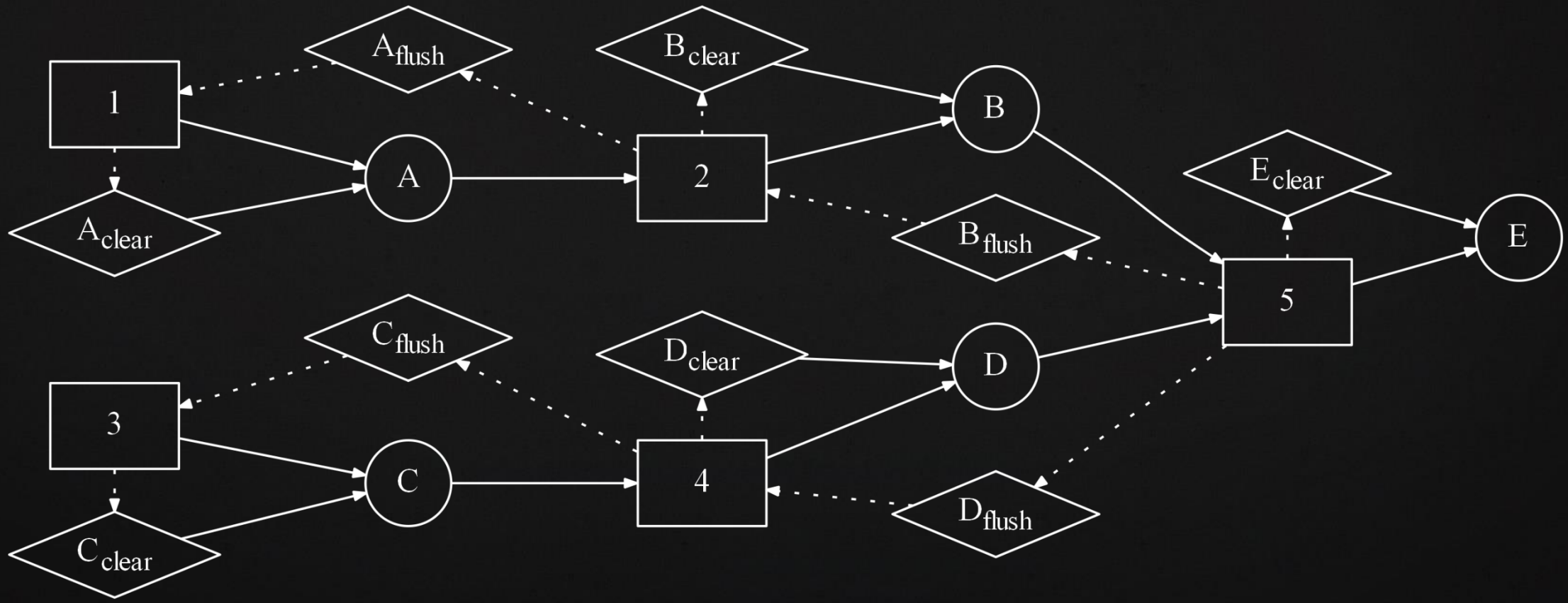
- Updated DAG, clears



FLUSH

MAKE SURE WE ALL AGREE

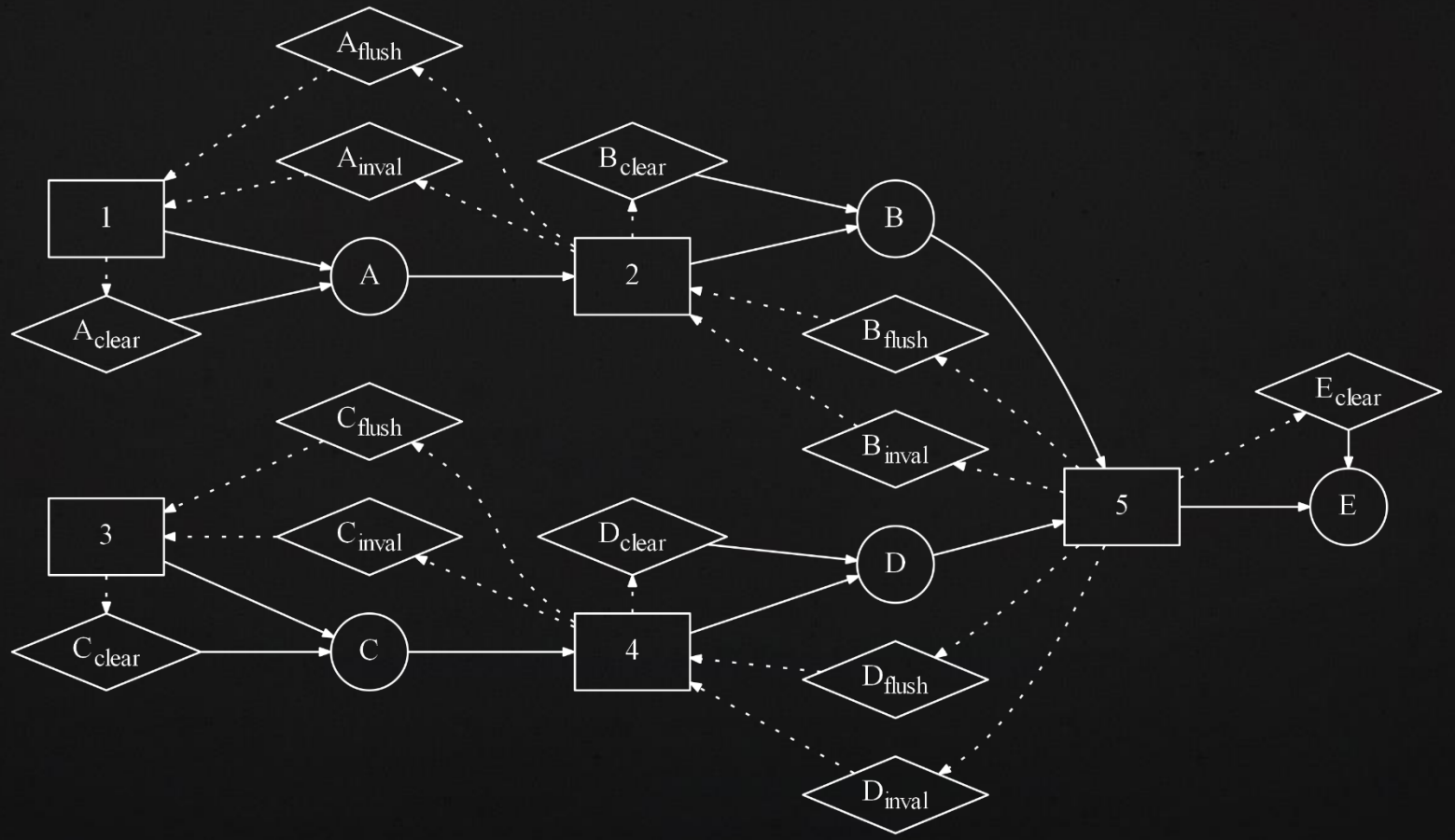
- Updated DAG, flushes



INVALIDATE

MAKE SURE WE REALLY AGREE

- Updated DAG, flushes, invalidation



PREDICTING THE FUTURE

IT'S EASY WHEN YOU KNOW HOW

- Renderpasses allow drivers to predict the future
  - Not really a prediction – you told it what you were going to do
  - Schedule clears, internal blits, cache operations, etc.
    - All done statically
    - When the renderpass is built
- "I can do that in the app, 'cuase I'm a 1337 haxxorz"
  - Well, no, you can't
  - Some of the internal driver operations aren't exposed in the API
  - Some are only needed on some hardware

LET'S GET CRAZY

DOUBLING DOWN ON THE NUTTY STUFF

- PSOs are built with respect to renderpasses
  - Each PSO knows which renderpass it will be used with, and in which subpass
  - Renderpass knows where subpass outputs go
  - Renderpass knows the format of all attachments
- If an output is not used, eliminate it
  - Reduce precision on outputs
  - Delete unused channels
- If an output is consumed directly
  - Fuse PSOs, specializing for the renderpass

SUMMARY

RECAP

- Renderpasses encapsulate data and execution flow
  - Driver can schedule internal work
  - Remove surprises at render time
  - Determine the fate of data early
- Many opportunities for GPU performance
  - Eliminate stalls and pipeline bubbles
  - Interleave internal operations with rendering
  - Optimize cache utilization
  - Choose formats and allocation strategies based on data flow

BEST PRACTICE

DO THIS

- Even a small renderpass of a couple of subpasses are good
  - Depth pre-pass, G-buffer render, lighting, post-process
- Dependencies aren't necessarily necessary
  - Multiple shadow map passes producing multiple outputs
- Fold stuff you're going to do anyway into the renderpass
  - Prefer load op clear over `vkCmdClearAttachment`
  - Prefer final layout on renderpass attachments over explicit barriers
  - Make liberal use of "don't care"
  - Perform MSAA resolves using resolve attachments

VULKAN FAST PATHS – BARRIERS & SYNC

DR. MATTHÄUS G. CHAJDAS (@NIV_Anteru)

AMD

BARRIERS

WHY DO WE NEED THEM

- Synchronization
  - Make sure writes have finished before reads start
  - Timing issues if missed
- Visibility
  - Caches are visible to other units
  - Partial results, flickering, etc.
- Decompression
  - Make sure formats match
  - Corruption if missed

BARRIERS

GET CODING NOW!

- vkCmdPipelineBarrier
- Points of interest:
  - Stage flags
  - The individual barriers
- Stage flags
  - Try to get the closest matching range
  - Compute while combining barriers
  - The smaller the range, the less units will idle - avoid TOP_OF_PIPE to BOTTOM_OF_PIPE

BARRIERS

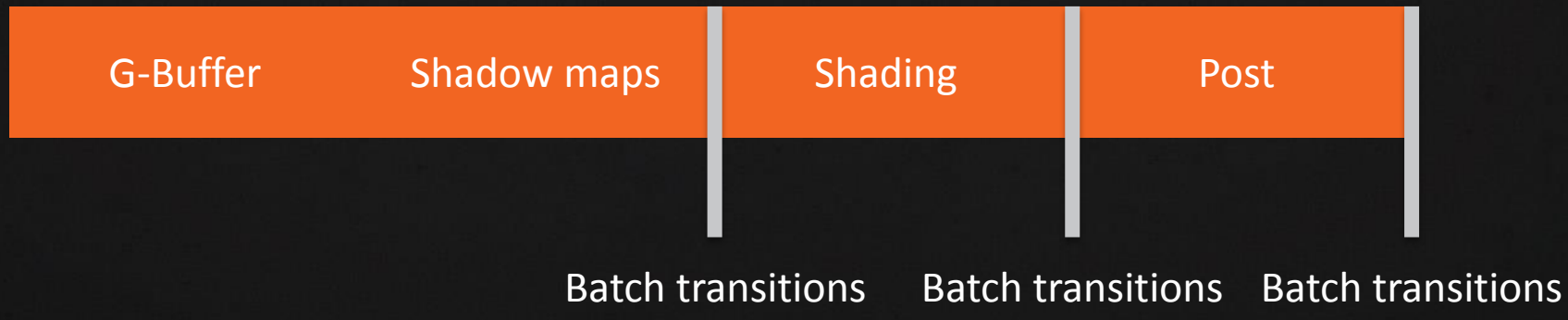
DON’T LET THEM STOP YOU

- For any of the barriers
  - Make sure to transition into the union of the read states
  - Or them together – avoid VK_ACCESS_MEMORY_READ_BIT
- Batch as many barriers as you can into one call
- Need to specify source/destination queue
- Place transition close to semaphore

BARRIER BATCHING

ALL FOR ONE

- If you can’t use render-passes **yet**, batch them at task boundaries
- Render passes are the **superior solution** for most barrier problems!



BARRIERS

ROAD TO 100% CORRECTNESS

- Avoid tracking per-resource state
  - –You don’t have that many resources that transition!
  - –State tracking makes batching hard
  - –Fragile
- Avoid transitioning everything – barriers have a cost!
  - –Cost often scales with resolution
  - –Cost changes between GPU generations

BARRIERS

TL;DR

- **As few barriers** as possible – don’t track per resource state
- Use **render passes** when possible
- Think about the **required** state

SYNCHRONIZATION

HOLD ON A MOMENT

- We have three synchronization primitives
  - Fences
  - Semaphores
  - Wait events
- Fences allow to synchronize GPU and CPU work
  - Frame sync
  - Protect frame resources with a fence
- Semaphore is a heavy-weight, cross queue sync
- Wait events are light-weight, in queue sync

SYNCHRONIZATON

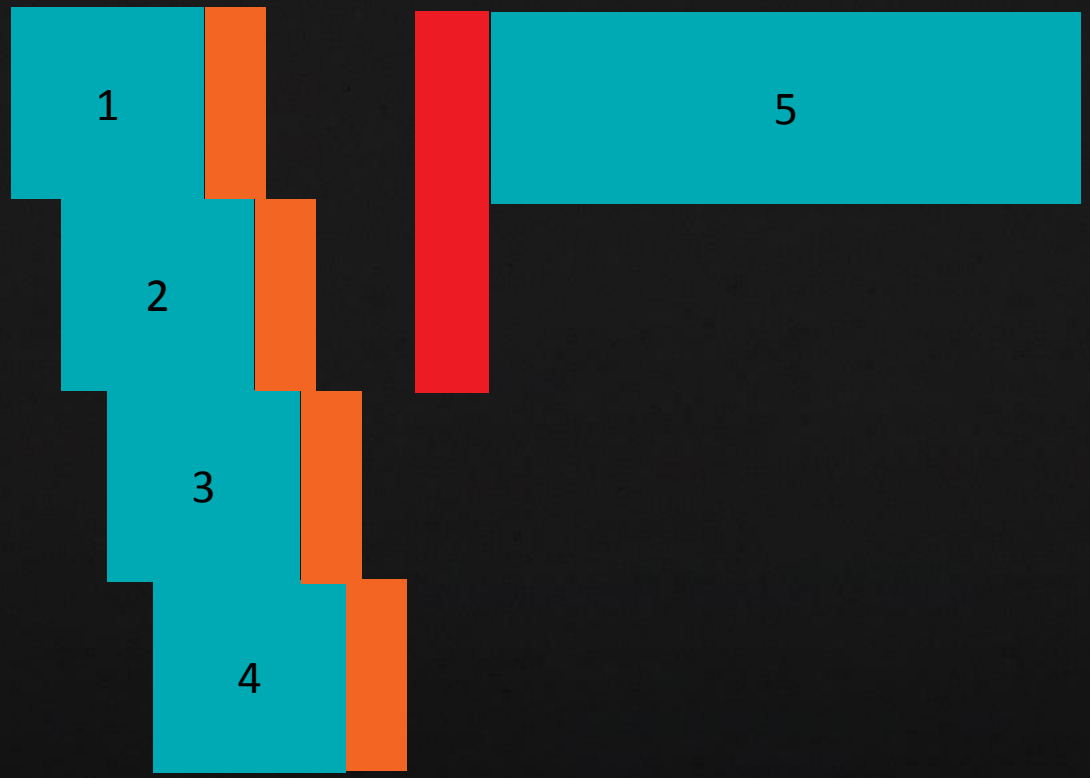
HOLD ON A SEC

- Wait events allow you to go wide

SYNCHRONIZATON

HOLD ON A SEC

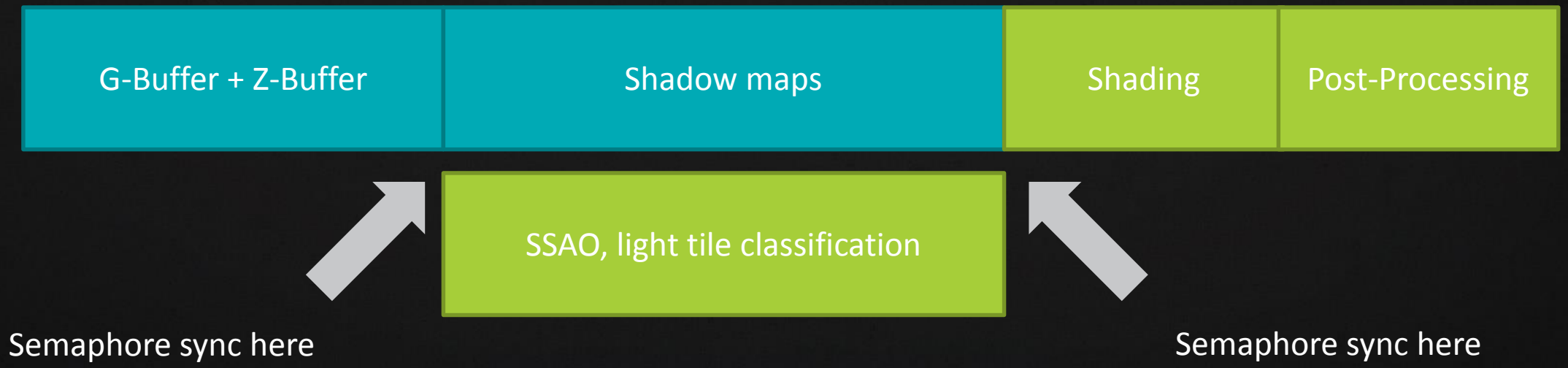
- Dispatch will continue while work executes
- Very cheap on GCN!



SYNCHRONIZATION

HOLD ON A MINUTE

- Semaphores are for cross-queue sync
- A couple per frame should suffice



SYNCHRONIZATION

TL;DR

- **Fence** once per frame to protect per-frame resources
- **Semaphores** when you go multi-queue
- **Wait events** if you want to go wide – especially on compute

THANK YOU

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION


AMD