Vulkan

Vulkan Pipeline

Pipeline Layout

Access to descriptor sets from a pipeline is accomplished through a pipeline layout. **Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object describing the complete set of resources that can be accessed by a pipeline.** The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

```
VkResult vkCreatePipelineLayout(
    VkDevice
 device,
    const VkPipelineLayoutCreateInfo*
pCreateInfo,
    const VkAllocationCallbacks*
 pAllocator,
    VkPipelineLayout*
pPipelineLayout);
```

MSAA

多重采样不同于许多后处理抗锯齿算法（如TAA、FXAA等），它可以完全在管线中工作，并且受到硬件支持。

Vulkan-Example [multisampling.cpp]

流程：

渲染模型 → 交换链显示
→ 将交换链图片保存到渲染附件
→ 对渲染附件做多重采样
→ 对多重采样后的图片进行降采样($Resolve$)
→ 重新渲染到交换链

硬件查询

Vulkan API对采样点最高定义为64，而现代GPU通常最多支持到8，所以在使用MSAA前，总共有两个硬件查询需要被发起：

1. 硬件最大可支持的采样点

```
1 VkPhysicalDeviceProperties pProperties;
2 vkGetPhysicalDeviceProperties(physicalDevice,
  &pProperties);
3 VkSampleCountFlags sampleCountSupported =
  pProperties.limits.*SampleCounts;
```

图1

> framebufferColorSampleCounts is a bitmask of VkSampleCountFlagBits indicating the color sample counts that are supported for all framebuffer color attachments with floating- or fixed-point formats. For color attachments with integer formats, see framebufferIntegerColorSampleCounts.

在得知硬件能力后，我们就可以为后续可能用到的`sampleCount`指定采样数了

```
VkSampleCountFlagBits getMaxUsableSampleCount()
{
    VkSampleCountFlags counts = std::min(deviceProperties.limits.framebufferColorSampleCounts,

 deviceProperties.limits.framebufferDepthSampleCounts);
    if (counts & VK_SAMPLE_COUNT_64_BIT) { return VK_SAMPLE_COUNT_64_BIT; }
    if (counts & VK_SAMPLE_COUNT_32_BIT) { return VK_SAMPLE_COUNT_32_BIT; }
    if (counts & VK_SAMPLE_COUNT_16_BIT) { return VK_SAMPLE_COUNT_16_BIT; }
    if (counts & VK_SAMPLE_COUNT_8_BIT)  { return VK_SAMPLE_COUNT_8_BIT; }
    if (counts & VK_SAMPLE_COUNT_4_BIT)  { return VK_SAMPLE_COUNT_4_BIT; }
    if (counts & VK_SAMPLE_COUNT_2_BIT)  { return VK_SAMPLE_COUNT_2_BIT; }
    return VK_SAMPLE_COUNT_1_BIT;
}
```

2. 硬件是否支持Sample Rate Shading（可选*）

```
VkPhysicalDeviceFeatures features;
vkGetPhysicalDeviceFeatures(physicalDevice, &features);
VkBool32 isSampleRateSupported = features.sampleRateShading;
```

准备渲染附件

为了给交换链上的图像进行多重采样，我们要先准备几个渲染附件用来临时保存它们。既然是临时使用，我们就可以将图片的使用方法定义为短时。

```
info.usage =
VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT |
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

> VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT specifies that the memory bound to this image will have been allocated with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT (see Memory Allocation for more detail). This bit can be set for any image that can be used to create a VkImageView suitable for use as a color, resolve, depth/stencil, or input attachment.

Spec中对短时用处的图片有特殊的定义，希望它们是被惰性分配内存的。

```
memAlloc.memoryTypeIndex = vulkanDevice->getMemoryType(memReqs.memoryTypeBits,
        VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT,
        &lazyMemTypePresent);
```

当然Device硬件的堆不一定支持这种内存方式，所以需要在查询内存要求进行查询。如果不支持的话，我们总是可以fallback回`DEVICE_LOCAL`的形式。

图2

最后要对多重采样的采样点数量进行定义，mip层级为1。

```
typedef struct VkImageCreateInfo {
    uint32_t                 mipLevels;
    VkSampleCountFlagBits    samples;
} VkImageCreateInfo;
```

准备降采样附件

图3

> Once a multisample image is built and rendered to, if it is to be viewed, it must be reduced down to a single-sample per-pixel. The process of downsampling a multisample image is called the "multisample resolve". In OpenGL, to perform a multisample resolve, you use a blit operation from a multisampled framebuffer to a single-sampled one. Note that such a resolve blit operation cannot also rescale the image or change its format.
>
> If you create a multisampled Default Framebuffer, the back buffer is considered multisampled, but the front buffer is not. So swapping the buffers is equivalent to doing a multisample resolve operation. [Multisampling]

管线内部添加多重采样

```
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineMultisampleStateCreateFlags flags;
    VkSampleCountFlagBits rasterizationSamples;  // sampleCount
    VkBool32 sampleShadingEnable;
    float minSampleShading;
    /*
     * 以下暂时不用
     */
    const VkSampleMask* pSampleMask;
    VkBool32 alphaToCoverageEnable;
    VkBool32 alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

Shader Storage Buffer Object (SSBO)

A **Shader Storage Buffer Object** is a Buffer Object that is used to store and retrieve data from within the OpenGL Shading Language.

SSBOs are a lot like Uniform Buffer Objects. Shader storage blocks are defined by Interface Block (GLSL)s in almost the same way as uniform blocks. Buffer objects that store SSBOs are bound to SSBO binding points, just as buffer objects for uniforms are bound to UBO binding points. And so forth.

The major differences between them are:

1. SSBOs can be much larger. The OpenGL spec guarantees that UBOs can be up to 16KB in size (implementations can allow them to be bigger). The spec guarantees that SSBOs can be up to 128MB. Most implementations will let you allocate a size up to the limit of GPU memory.
2. SSBOs are writable, even atomically; UBOs are uniforms. SSBOs reads and writes use incoherent memory accesses, so they need the appropriate barriers, just as Image Load Store operations.
3. SSBOs can have variable storage, up to whatever buffer range was bound for that particular buffer; UBOs must have a specific, fixed storage size. This means that you

can have an array of arbitrary length in an SSBO (at the end, rather). The actual size of the array, based on the range of the buffer bound, can be queried at runtime in the shader using the length function on the unbounded array variable.

4. SSBO access, all things being equal, will likely be slower than UBO access . SSBOs are generally accessed like buffer textures, while UBO data is accessed through internal shader-accessible memory reads. At the very least, UBOs will be no slower than SSBOs.

Compute Shaders have the following built-in input variables.

阅读看一下这个！！！！！！！

```
in uvec3 gl_NumWorkGroups;
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;
in uvec3 gl_GlobalInvocationID;
in uint  gl_LocalInvocationIndex;
```

- gl_NumWorkGroups
  This variable contains the number of work groups passed to the dispatch function.
- gl_WorkGroupID
  This is the current work group for this shader invocation. Each of the XYZ components will be on the half-open range [0, gl_NumWorkGroups.XYZ).
- gl_LocalInvocationID

This is the current invocation of the shader *within* the work group. Each of the XYZ components will be on the half-open range [0, gl_WorkGroupSize.XYZ).

- gl_GlobalInvocationID

  This value uniquely identifies this particular invocation of the compute shader among *all* invocations of this compute dispatch call. It's a short-hand for the math computation:

```
gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;
```

- gl_LocalInvocationIndex

  This is a 1D version of gl_LocalInvocationID. It identifies this invocation's index *within* the work group. It is short-hand for this math computation:

VKUnmap

VK_DESCRIPTOR_TYPE

首先，我们大致看一下Vulkan总共提供有哪些描述符种类，

```
typedef enum VkDescriptorType {
    VK_DESCRIPTOR_TYPE_SAMPLER = 0,
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER = 1,
```

```
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE = 2,
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE = 3,

    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER = 4,
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER = 5,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER = 6,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER = 7,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC =
8,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC =
9,

    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT = 10,

    /* Extensions */
    // Provided by VK_EXT_inline_uniform_block
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT
= 1000138000,
    // Provided by VK_KHR_acceleration_structure

 VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR =
1000150000,
    // Provided by VK_NV_ray_tracing
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV
= 1000165000,
    // Provided by
VK_VALVE_mutable_descriptor_type
    VK_DESCRIPTOR_TYPE_MUTABLE_VALVE =
1000351000,
} VkDescriptorType;
```

VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE

Sampled Image需要配合Sampler一起使用来达到Combined Image Sampler的作用。使用Combined Image Sampler时，我们需要将`VkSampler`和`VkImageView`一起绑定到描述符`VkWriteDescriptorSet.pImageInfo`中，也就是说，此时我们是将Sampler和ImageView打包绑定到一起发送给Fragment Shader中进行使用。

相反，Sampled Image和Sampler都是独立的发送给着色器，我们可以在着色器中自由的使用管线中绑定的Sampled Image和Sampler。举个栗子，假如我们想要在一个应用中对同一副图片每一帧都进行不同的过滤方式。

1. 方式一：使用`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`，我们先要在应用里创建一个`VkSampler`和一个`VkImageView`并将他们绑定到描述符集上，打包发送给Fragment Shader。在下一帧时，再更新`VkSampler`的描述符，重新写入管线。
2. 方式二：使用`VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`和`VK_DESCRIPTOR_TYPE_SAMPLER`，我们创建一个`VkImageView`和多个`VkSampler`，将他们分别绑定到Fragment Shader，接下来我们只需要每一次在Shader中切换绑定到Sampled Image上Sampler即可。

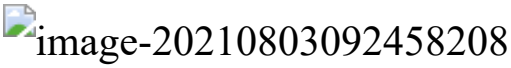
Multithreading

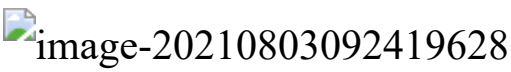
API端的并行渲染的主要思想不在于GPU端进行了多少并行计算等处理（与GPU端并行渲染相异），而在于并行录制指令。

Vulkan中主要两种并行录制指令的方式：

1. 针对每一个线程单独创建Command Buffer并录制指令，主要针对于桌面级GPU
2. 针对同一个Render Pass使用Secondary Command Buffer录制指令，主要针对于
    1. 移动级（TBR/TBDR）GPU
    2. 受制于Render Pass，需要在同一个Render Pass内完成渲染

Memory





Run out of `DEVICE_LOCAL` memory (VRAM) and try `vkAllocateMemory`

Both are correct:

1. `VK_ERROR_OUT_OF_DEVICE_MEMORY` : allocation fails; application must handle the failure
2. `VK_SUCCESS` : application succeeds; some blocks are silently migrated to system memory (accessing blocks degrades GPU performance)

depending on driver/hardware

Implicit resources which need memory:

- Swap chains
- Command Buffers
- Descriptors
- Shaders / PSOs
- Query Results

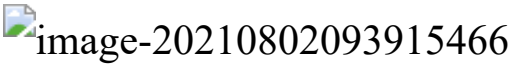
Hence, use `VkMemoryHeap::size` applying "informed adjustments"

Flags	Hack
DEVICE_LOCAL	VkMemoryHeap::size * 0.8f
DEVICE_LOCAL | HOST_VISIBLE	VkMemoryHeap::size * 0.66f

Aliasing

[Double check]

- As the resolution gets larger, the render targets follows
- As resources are transient, aliasing can be a solution to keep render target/UAV memory in check
- Better assuming it contains garbage
- > 50% memory are saved in some titles [ODonnel17]



Transfers

> Transfer Queue is designed to transfer resources from host to `DEVICE_LOCAL` memory. However, in the case of `DEIVCE_LOCAL` to `DEVICE_LOCAL`, because of its nature of asynchronism, copying in transfer queues might introduce some of latencies. On the other hand, pipeline

> copy operations in Graphics/Compute queue might be faster and provide with zero latency.
>
> - Need it now - Graphics/Compute queue
> - It can wait - Transfer queue

`VK_QUEUE_TRANSFER_BIT` can be used to copy a resource from host memory to `DEVICE_LOCAL` memory

- In AMD devices, transfer queue is mapped to the DMA engine (hardware) asynchronously communicating with the rest of the chip
- Fastest way to copy across PCIe bus
- Better do such things way before need them (transferred resources) on graphics/compute queue because of the nature of asynchronism
- In contrast, peak transfer rates of Graphics/Compute are probably faster, but it needs to monitor the GPU clock for resource copy and etc, so clogging here.

Mapping

- Having entire memory block persistently mapped is generally ok, which means no longer any need to unmap before using stuff on GPU
- Keeping large blocks definitely cause stability/performance issues

Creation

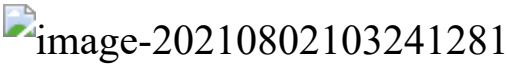
- `VkImageCreateInfo.sharingMode` : VK_SHARING_MODE_CONCURRENT triggers compression, go for VK_SHARING_MODE_EXCLUSIVE and use explicit queue family ownership barriers.
- `VkImageCreateInfo.initialLayout:` For the initalLayout, it must be either VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED. Besides, always prefre VK_IMAGE_LAYOUT _ * _ OPTIMAL state rather than VK_IMAGE_LAYOUT_GENERAL
- `VkImageCreateInfo.tiling` : VK_IMAGE_TILING_OPTIMAL is more optimal than VK_IMAGE_TILING_LINEAR
- `VkImageCreateInfo.usage` : Avoid setting to much bits on it, it's a great way to confuse the driver into flushing more caches, and draining from GPU

Resource Sizes

- Don't cache the results when querying sizes - querying the size required for two identical resources might not return the same results.
- Make sure query each resource for it's own specific requirements

Allocations

- Better not mixing large and small allocations
- Consider routing allocations to different blocks of memory based on their sizes
- Pool larger allocations in one block, small allocations in another block - tighter packing
- Reduces small fragmentations not removes them. But it is statistically more likely to get free space in either memory pool.



Vulkan Synchronizations

目录

> 分析顺序将从同步粒度从粗到细逐步深入

- Fence
- Semaphore

- Barrier
- Subpass Dependency
- Event

Vulkan Synchronizations

目录

> 分析顺序将从同步粒度从粗到细逐步深入

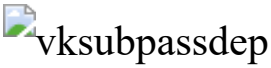
- Fence
- Semaphore
- Barrier
- Subpass Dependency
- Event

Barrier

Subpass Dependency

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

- `srcSubpass` : which subpass I am depending on
- `dstSubpass` : who I am (`srcSubpass` almost always has the lower index than `dstSubpass` does, otherwise UNDEFINED)
- `srcStageMask` & `dstStageMask` : which stages everything depending on



```
VkSubpassDependency sp = {
    .srcSubpass = 0,
    .dstSubpass = 1,
    .srcStageMask = LATE_FRAGMENT_TESTS,
    .dstStageMask = EARLY_FRAGMENT_TESTS,
    .srcAccessMask =
DEPTH_STENCIL_ATTACHMENT_WRITE,
    .dstAccessMask =
DEPTH_STENCIL_ATTACHMENT_READ,
    .dependencyFlags = BY_REGION
}
```

A pipeline barrier in a subpass will create a dependency between commands within a subpass. Subpass dependencies create dependencies between subpasses. External subpass dependencies create dependencies between a subpass and the commands before/after the render pass.

If subpass 1 cannot start its execution until subpass 0 has finished (and therefore, there is a dependency between them), then any commands in subpass 1 can assume that subpass 0 is done. This includes barriers. So this makes dependencies *transitive*; the stuff after the barrier in subpass 1 can assume that subpass 0 is finished because *everything* in subpass 1 can make that assumption. Similarly, commands in a subpass (like barriers) will depend on any external dependencies which the subpass directly or indirectly depends on.

> Now, because dependencies are be based on particular stages, transitivity only applies when the chain of dependencies include stages that actually depend on each other.
>
> The only implicit dependencies are external to the renderpass.

Implicit Dependencies:

```
// or other way around
{
    srcSubpass = VK_SUBPASS_EXTERNAL;
    dstSubpass = 0;
}
```

Explicit Dependencies:

```
{
    srcSubpass = 0;
    dstSubpass = 1;
}
```

Vulkan Synchronization Model

Vulkan通过引入执行层面 (Execution Dependency) 和资源层面 (Memory Dependency) 两方面依赖来保证数据的依赖关系。

Execution Dependency

执行层面的依赖主要维护提交的GPU指令之间的依赖，比如

```
vkCmdDispatch();

vkCmdpipelineBarrier(
    ...
    VK_PIPELINE_STAGE_XXX_SHADER_BIT,
    VK_PIPELINE_STAGE_XXX_SHADER_BIT,
    ...
)

vkCmdDispatch();
```

Memory Dependency

资源层面的依赖主要维护数据的可用性、可依赖性

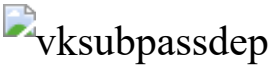
Semaphore

Barrier

Subpass Dependency

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

- `srcSubpass` : which subpass I am depending on
- `dstSubpass` : who I am (`srcSubpass` almost always has the lower index than `dstSubpass` does, otherwise UNDEFINED)
- `srcStageMask` & `dstStageMask` : which stages everything depending on



```
VkSubpassDependency sp = {
    .srcSubpass = 0,
    .dstSubpass = 1,
    .srcStageMask = LATE_FRAGMENT_TESTS,
    .dstStageMask = EARLY_FRAGMENT_TESTS,
    .srcAccessMask =
DEPTH_STENCIL_ATTACHMENT_WRITE,
    .dstAccessMask =
DEPTH_STENCIL_ATTACHMENT_READ,
    .dependencyFlags = BY_REGION
}
```

A pipeline barrier in a subpass will create a dependency between commands within a subpass. Subpass dependencies create dependencies between subpasses. External subpass dependencies create dependencies between a subpass and the commands before/after the render pass.

If subpass 1 cannot start its execution until subpass 0 has finished (and therefore, there is a dependency between them), then any commands in subpass 1 can assume that subpass 0 is done. This includes barriers. So this makes dependencies *transitive*; the stuff after the barrier in subpass 1 can assume that subpass 0 is finished because *everything* in subpass 1 can make that assumption. Similarly, commands in a subpass (like barriers) will depend on any external dependencies which the subpass directly or indirectly depends on.

> Now, because dependencies are be based on particular stages, transitivity only applies when the chain of dependencies include stages that actually depend on each other.
>
> The only implicit dependencies are external to the renderpass.

Implicit Dependencies:

```
// or other way around
{
    srcSubpass = VK_SUBPASS_EXTERNAL;
    dstSubpass = 0;
}
```

Explicit Dependencies:

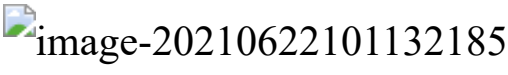
```
{
    srcSubpass = 0;
    dstSubpass = 1;
}
```

Vulkan Session

This is the note taking from all sorts of Vulkan Talks in a perspective of API view, including GDC and SIGGRAPH.

Vulkan Memory

Corherent/Mappable/Coherent Allocations only shall be defined if they are needed.



When you call reset command, the memory goes back to the upper level pool.

```
VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;

VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
}
```

Allocate a bunch of command buffers once. Don't do one command buffer once a time.

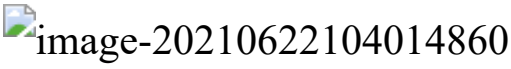
> Secondary Command Buffers needs a little bit state information which helps seamless translations.

If you want to have queries active while exectuing the second command buffers.

```
pInheritantceInfo.occlusionQueryEnable
pInheritantceInfo.queryFlags
pInheritantceInfo.pipelineStatistics
```

If you set `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`, you cannot change the state of the command buffer in the middle. That is saying if you cannot re-begin it while it currently begins, nor you cannot change it

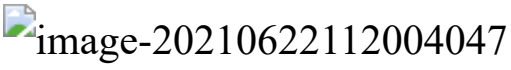
While the command buffer is in the `pending execution` state, you cannot change the state of it nor free it.



Use semaphore if you want sychronzation between command buffers.

If you don't need fence in submission, you can put it in subsequent submission

Gather a batch of command buffers and submit them altogether. Yet, there are the trade off between the time of gathering batches and the time vulkan is waiting for next instructions.



Command Buffer Recycling - reset can either recycle or release resources

- Recycling options:
  - Reset Command Pool - reset all command buffers in pool
  - Reset Command Buffer - reset single command buffer
- Use VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
- Handling memory growth:
  - Destroy command buffers no longer needed
  - Heuristic to destroy command buffers0

Command Buffer Batching

- `vkQueueSubmit` implies a flush
- Important to batch command buffers
   image-20210622155456032
- methodology
  - Set up a threshold for the maximum number of command buffers
  - Issue single submit with all batched command buffers

- Since the latency (time between CPU submits and GPU actually runs instructions) is so low, ones won't worry too much about GPU to be idle at all.

Redundant Call Filtering

- Vulkan Drivers may (should) not filter calls as the Vulkan API is supposed to be as thin as possible
- Global dirty state checking for calling only if absoutely have to （OpenGL also adopts this concept while most apps rely on driver to do this for them)

Updating Descriptors

- In shaders, organize descriptor sets by update frequency
  - treat textures as descriptor sets, bake it upfront and never change
  - treat uniform buffers as dynamic offset, only access uniform buffers by varying the offset
- Bake descriptors sets up front
- Use compatible pipeline layouts
  - keep pipeline layout from getting too much unique

Pipeline Creation

- vkCreateShaderModule is relatively fast - loadsin the SPIR-V, no heavy compilation
- vkCreateGraphicsPipelines is expensive - driver performs shader compile

Pipeline Cache

- Avoid unnecessary shader compiles (how)
  - Driver de-duplicates (probablly done)
  - Only driver knows when recompile is needed based on state
  - Pipeline cache should contian only unique pipelines
- Allow compilation on multiple threads (multi-threading may not achievable)
  - merge using `vkMergePipelineCaches`

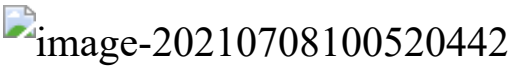
Hardware limits

- A lot of Vulkan variables might have the maximum number of instances, they might require you to query the hw before or figure out some kinds of pool mechanisms.

OpenGL vs Vulkan

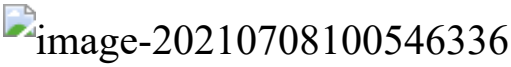
> Core conception: Don't make driver to guess, make/pass more information upfront.

Command Buffers



OpenGL is running in a serieral way, whilst Vulkan is running in an asynchronous way. That is saying, OpenGL almost has a similar GPU timeline to the CPU timeline. But Vulkan don't.

> Migrate OpenGL to Vulkan isn't just about one-to-one update. But really to rip out the application and rethink how the application should work to take the advantage of such new APIs.



Pipeline

The one key part about pipeline in Vulkan is its mutibility (Dynamic State?). OpenGL might have to wait for all the state are avilable to be able to compile the pipeline. Consider the scenario, where you are going to do `glLinkProgram(); glCompileShader()`, while your shader are waiting for your input (uniforms, etc). The shader might not get to be compiled or your pipeline might not get to be compiled until its state are satisfied per se.

The driver has to do the guessing, because there is on way that applications can tell OpenGL they aren't doing somerthing. The driver, thus, results in considering wrost case every time.

Layer

When Vulkan is first designed, there are layers per instance or per device. The user decide the extension that goes to which device or which instance. Later on, the concept of device layer is then deprecated. Hence, layers enabled in instance level is also enabled for device level you created using that instance.