VULKAN AND NVIDIA: THE ESSENTIALS

Tristan Lorach Manager of Developer Technology Group, NVIDIA US

7/25/2016
ANALOGY ON GRAPHIC APIS

(getting ready for my 7 years old son’s questions on my job...)

Car Toy
Lego Kit
Derby Kit
Analogy
Different Valid Approaches

(booring...) (cool... Messes-up the bedroom) (adult supervision required!)

Fixed-function OpenGL Modern AZDO OpenGL with Programmable Shaders Vulkan
WHAT IS VULKAN?

...It’s a modern API

- Designed and maintained by Khronos Group
- Designed for high performance on rendering and compute
- [Extremely] low level: no more “baby-sitting” from our driver
  - Manage yourself memory, resource updates; batching; scheduling...
- [Extremely] verbose: Lots of structures to fill with parameters
- close to DX12 design...
- Opposite of OpenGL: Multi-threading friendly: Vulkan will especially shine if multi-threading used
- But still generic enough to work on many HW vendors & platforms
Is your graphics work CPU bound?  yes  Can your graphics creation be parallelized?  yes

Your graphics platform is fixed  yes  You’ll do whatever it takes to squeeze out Max perf.  yes

You put a premium on avoiding hitches  yes  You can manage your graphics resource allocations  yes

Vulkan friendly
Beneficial Vulkan Scenarios

- Is your graphics work CPU bound?
  - yes

- Can your graphics creation be parallelized?
  - yes

Tired with OpenGL (state-machine) or even D3D?
  - Kinda... (it’s a Yes)

Want to learn new stuff?
  - You put a premium on avoiding hitches
  - You can manage your graphics resource allocations
  - No sleep?
  - Alright... (Yes)

Vulkan friendly
Unlikely to Benefit
Scenarios to Reconsider Coding to Vulkan

1. Need for compatibility to pre-Vulkan platforms
2. Heavily GPU-bound application
3. Heavily CPU-bound application due to non-graphics work
4. Single-threaded application, unlikely to change
5. App can target middle-ware engine, avoiding 3D graphics API dependencies
   • Consider using an engine targeting Vulkan, instead of dealing with Vulkan yourself

Good News in any case: NVIDIA OpenGL driver is great and will always be there!
BIG PICTURE - OPENGL CASE

Application

OpenGL Commands

OpenGL resources

Application

OpenGL Commands

OpenGL resources

Graphics pipeline

States

Resources

Dependencies

Heap

cmd bundles

memory

Element buffer (EBO)

Draw Indirect Buffer

Vertex Buffer (VBO)

Uniform Block

Texture Fetch

Image Load/Store

Atomic Counter

Shader Storage

FBO resources (Textures / RB)

Tr. Feedback buffer

GPU

FIFO

cmds

Front-End (decoder)

Vertex Puller (IA)

Vertex Shader

TCS (Tessellation)

Tessellator

TES (Tessellation)

Geometry Shader

Transform Feedback

Rasterization

Fragment Shader

Per-Fragment Ops

Framebuffer
BIG PICTURE - VULKAN

**Application**
- Cmd-buffers / queues
- Pipeline States
- Rend Passes
- Descriptor Sets
- Resources
- Dependencies

**OpenGL Driver**
- Fewer translation, Validation checks And internal mgt
- Minimal memory management

**Memory**
- Element buffer (EBO)
- Draw Indirect Buffer
- Vertex Buffer (VBO)
- Uniform Block
- Texture Fetch
- Image Load/Store
- Atomic Counter
- Shader Storage
- FBO resources (Textures / RB)
- Tr. Feedback buffer

**GPU**
- Front-End (decoder)
- Vertex Puller (IA)
- Vertex Shader
- TCS (Tessellation)
- Tessellator
- TES (Tessellation)
- Geometry Shader
- Transform Feedback
- Rasterization
- Fragment Shader
- Per-Fragment Ops
- Framebuffer
VULKAN COMPONENTS

Heap

Memory

Heap

Memory

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

... ...

Cmd.Buffer Pool

Command-buffer

Barrier synchronization

Begin Render-Pass

Bind Graphics-pipeline

Set misc. dynamic states

Bind Vertex/Idx Buffer(s)

Update Buffer

Bind Descriptor-Set(s)

Draw...

Execute Commands

End Render-Pass

Instance

Device

Queue
VULKAN COMPONENTS

- Instance
- Device(s)
- Queue(s)
- Heap
- Memory
- Image View
- Framebuffer
- Render-Pass
- Graphics pipeline
- Buffer
- Descriptor-Set
- DescriptorSet Pool
- Image View
- Image
- Buffer
- Sampler
- Command-buffer Pool
- Command-buffer
  - Barrier synchronization
  - Begin Render-Pass
  - Bind Graphics-pipeline
  - Set misc. dynamic states
  - Bind Vertex/Idx Buffer(s)
  - Update Buffer
  - Bind Descriptor-Set(s)
  - Draw...
  - Execute Commands
  - End Render-Pass
VULKAN OBJECTS: DEVICE

Instance ~⇔~ OpenGL Context

Instance-Layers
- Intercepting API calls for misc. purposes
- Many layers available (api-dump; core/std/parms validation; screenshot...)

Instance-Specific Extensions
- KHR_Surface (for Swap-chains)
- EXT_debug_report
- ...

Exposes some Devices...
VULKAN OBJECTS: DEVICE

VkPhysicalDevice
- Capabilities
- Memory Management
- Queues
- Objects
  - Buffers
  - Images
  - Sync Primitives

Device(s)
NVIDIA’S VULKAN CAPABILITIES

Properties listed from Physical Device

NVIDIA is almost full featured

  Top to bottom: from GeForce, Quadro down to Tegra

Check http://vulkan.gpuinfo.org/listreports.php
# NVIDIA’S VULKAN CAPABILITIES

## GeForce GTX 980

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphaToOne</td>
<td>true</td>
</tr>
<tr>
<td>depthWriteTessCoord</td>
<td>true</td>
</tr>
<tr>
<td>depthWrite</td>
<td>true</td>
</tr>
<tr>
<td>depthWriteIndexBits</td>
<td>true</td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
</tr>
<tr>
<td>dualSrcBlend</td>
<td>true</td>
</tr>
<tr>
<td>fillModeSolid</td>
<td>true</td>
</tr>
<tr>
<td>fragmentStoresAndAtomics</td>
<td>true</td>
</tr>
<tr>
<td>fullDrawIndirect32</td>
<td>true</td>
</tr>
<tr>
<td>geometryShader</td>
<td>true</td>
</tr>
<tr>
<td>imageCubeArray</td>
<td>true</td>
</tr>
<tr>
<td>independentBlend</td>
<td>true</td>
</tr>
<tr>
<td>inheritedQueries</td>
<td>true</td>
</tr>
<tr>
<td>largePoints</td>
<td>true</td>
</tr>
<tr>
<td>logicOp</td>
<td>true</td>
</tr>
<tr>
<td>multiDrawIndirect</td>
<td>true</td>
</tr>
<tr>
<td>multiViewport</td>
<td>true</td>
</tr>
<tr>
<td>occlusionQueryPrecise</td>
<td>true</td>
</tr>
<tr>
<td>pipelineStatisticsQuery</td>
<td>true</td>
</tr>
<tr>
<td>robustBufferAccess</td>
<td>true</td>
</tr>
<tr>
<td>sampleRateShading</td>
<td>true</td>
</tr>
<tr>
<td>samplerAnisotropy</td>
<td>true</td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderCullDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderFloat32</td>
<td>true</td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>false</td>
</tr>
<tr>
<td>shaderInt8</td>
<td>false</td>
</tr>
</tbody>
</table>

## Tegra X1 & K1

<table>
<thead>
<tr>
<th>Feature</th>
<th>NVIDIA NVIDIA Tegra X1</th>
<th>NVIDIA NVIDIA Tegra K1</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>361.0.0 (1.0.2)</td>
<td>361.0.0 (1.0.2)</td>
</tr>
<tr>
<td>api</td>
<td>android 6.0 (arm)</td>
<td>android 6.0 (arm)</td>
</tr>
<tr>
<td>alphaToOne</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>depthBiasClamp</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>depthBounds</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>depthClamp</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>dualSrcBlend</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fillModeSolid</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fragmentStoresAndAtomics</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>fullDrawIndirect32</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>geometryShader</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>imageCubeArray</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>independentBlend</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>inheritedQueries</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>largePoints</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>logicOp</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>multiDrawIndirect</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>multiViewport</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>occlusionQueryPrecise</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>pipelineStatisticsQuery</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>robustBufferAccess</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>sampleRateShading</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>samplerAnisotropy</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderCullDistance</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderFloat32</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderInt8</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>shaderInt16</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

- **shaderResourceMinLod**: true, false
- **shaderResourceResidency**: true, false
- **shaderSampleImageArrayDynamicIndexing**: true, false
- **shaderStorageBufferArrayDynamicIndexing**: true, false
- **shaderStorageImageArrayDynamicIndexing**: true, false
- **shaderStorageImageExtendedFormats**: true, false
- **shaderStorageImageMultisample**: true, false
- **shaderStorageImageMultisampleWithFormat**: true, false
- **shaderStorageImageWriteWithoutFormat**: true, false
- **shaderTessellationAndGeometryIndexSize**: true, false
- **shaderUniformBufferArrayDynamicIndexing**: true, false
- **sparseBinding**: true, false
- **sparseResidency4xSamples**: true, false
- **sparseResidency8xSamples**: true, false
- **sparseResidency16xSamples**: true, false
- **sparseResidency32xSamples**: true, false
- **sparseResidency64xSamples**: true, false
- **sparseResidency128xSamples**: true, false
- **sparseResidencyBuffer**: true, false
- **sparseResidencyImage2D**: true, false
- **sparseResidencyImage3D**: true, false
- **tessellationBlend**: true, false
- **textureCompressionASTC_LDR**: true, false
- **textureCompressionBC**: true, false
- **textureCompressionETC2**: true, false
- **variableMultisampleRate**: true, false
- **vertextPipeAndShadersAndAtomics**: true, false
- **vertextPipeAndShadersAndAtomics**: true, false
Command queue was hidden in OpenGL Context... now explicitly declared

Multiple threads can submit work to a queue (or queues)!

Queues accept GPU work via CommandBuffer submissions

Few operations available around Queues: “submit work” and “wait for idle”

Queue submissions can include sync primitives for the queue to:

Wait upon before processing the submitted work

Signal when the work in this submission is completed

Queue “families” can accept different types of work, e.g.

NVIDIA exposes 2 families: 1+16 Queues

16 for all available types of work

1 for transfer operations only (Copy Engine)
VULKAN COMPONENTS

Heap

Memory

Device

Queue

Heap

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

...
SYNCHRONIZATION

**events and barriers**

used to synchronize work within a command buffer or sequence of command buffers submitted to a single queue

**semaphores**

used to synchronize work across queues or across coarse-grained submissions to a single queue

**fences**

used to synchronize work between the device and the host.
**COMMAND-BUFFERS**

**Vulkan Rendering ➔ Command-Buffers**

Close to what GPU will get at Front-End (FIFO)

Minor *translation & optimization* from the Driver prior to sending to the GPU

Each can be created either for **one shot** or for **multiple frames/submissions**

Cannot Cmd-Buffers from GPU (command-lists can): API calls to `vkCmd...()` between Begin & End

**Multi-threading** friendly (!)

**Primary** Cmd-Buffer can call many **2ndary** Cmd-Buffers
COMMAND-BUFFERS AND MULTI-THREADING

Main thread (Busy)
- Game Work
- Thread Coordination
  - cmd. Buffer Pool
  - Create 1-ary Cmd Buffer
  - Collect
  - 1-ary Cmd calls 2-dary ones
  - Submit to Q
  - Swapping

Thread Coordination

Thread 1 (Busy)
- Update Work
  - cmd. Buffer Pool
  - Create 2-dary Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

Thread 2 (Busy)
- Update Work
  - cmd. Buffer Pool
  - Create 2-dary Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

Thread 3
- Update Work
  - cmd. Buffer Pool
  - Create 2-dary Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

Thread 4
- Update Work
  - cmd. Buffer Pool
  - Create 2-dary Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

Command Buffer Pool local to the thread To prevent conflicts in concurrent access
COMMAND BUFFER THREAD SAFETY

Must not recycle a CommandBuffer for rewriting until it is no longer in flight (In flight == GPU still consuming it on its side)

But we can’t flush the queue each frame: would break parallelism!

VkFences can be provided with a queue submission to test when a command buffer is ready to be recycled

GPU Consumes Queue

Fence A Signaled to App

Fence A

Fence B

CommandBuffer

CommandBuffer

CommandBuffer

CommandBuffer

CommandBuffer

CommandBuffer

App Submissions to the Queue

Rewrite command buffer
THREADS AND COMMAND POOLS

Threads can have more than 1 Command Pool

**Ring-buffer**: One Command-Pool per Frame

when the frame is no longer in flight (Using Fences):

simply reset the whole Pool

<table>
<thead>
<tr>
<th>Frame N-2</th>
<th>Frame N-1</th>
<th>Frame N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
<td>![CommandPool](Command Buffer) ![Command Buffer](Command Buffer)</td>
<td>![CommandPool](Command Buffer) ![Command Buffer](Command Buffer)</td>
</tr>
<tr>
<td><strong>Thread 2</strong></td>
<td>![CommandPool](Command Buffer) ![Command Buffer](Command Buffer)</td>
<td>![CommandPool](Command Buffer) ![Command Buffer](Command Buffer)</td>
</tr>
</tbody>
</table>
**GRAPHICS PIPELINE**

Snapshot of all States

Including **Shaders**

Pre-compiled & Immutable

**Ideally:** done at Initialization time

Ok at render-time *if* using the Pipeline-Cache

Prevents validation overhead during rendering loop

Some Render-states can be excluded from it: they become “Dynamic” States

---

Pipe (Line cache)

Descriptor-set Layout

Graphics pipeline

Optional Dynamic States

- Viewport
- Scissor
- Blend const
- Stencil Ref
- Depth Bounds
- Depth Bias

- depthClipEnable
- rasterizerDiscardEnable
- fillMode
- cullMode
- frontFace
- depthBiasEnable
- depthBias
- depthBiasClamp
- slopeScaledDepthBias
- lineWidth

Shader Module

- Shader Stage
- Vertex Input
- Tess. State
- Viewport State

---

Shader Module

- Shader Stage
- Vertex Input
- Tess. State
- Viewport State

---

Pipeline layout

Descriptor-set Layout

Graphics pipeline

Optional Dynamic States

- Viewport
- Scissor
- Blend const
- Stencil Ref
- Depth Bounds
- Depth Bias

- depthClipEnable
- rasterizerDiscardEnable
- fillMode
- cullMode
- frontFace
- depthBiasEnable
- depthBias
- depthBiasClamp
- slopeScaledDepthBias
- lineWidth
GRAPHICS PIPELINE

Graphics Pipeline must be consistent with shaders

No “introspection”, so everything known & prepared in advance

Vertex Input:

tells how Attributes: Locations are attached to which Vertex Buffer at which offset

Pipeline Layout:

Tells how to map Sets and Bindings for the shaders at each stage (Vtx, Fragment, Geom...)

GLSL Code

```glsl
layout(std140, set=0, binding=0) uniform A { ... };
layout(std140, set=0, binding=1) uniform B { ... };
layout(std140, set=1, binding=2) uniform C { ... };
...
layout(location=0) in vec3 pos;
layout(location=1) in vec3 N;
void main() { ...
```
BUFFERS

Highly Heterogenous. Most often used for:

- Index/Vertex Buffers
- Uniform Buffers (Matrices, material parameters...)

**Vulkan Object: Must be bound to some Device Memory**

- Can be **CPU accessible** memory (mappable)
- Can be **CPU cached**
- Can be **GPU accessible** only: need a “Staging Buffer” to write into it

But most Efficient

(More on Device Memory later...)
COMMAND-BUFFERS: UPDATE/PUSH CONSTANTS

2 more ways to update constants/uniforms for Shaders from the Command-Buffer

**Update-Buffer**: prior to Render-Pass: can target any Buffer bound by Descriptor Sets

```cpp
class objectBuffer {
    mat4 matrixObject;
    vec4 diffuse;
} object;
```

**Push-Constants**: targets a dedicated section in GLSL/SpirV

```cpp
layout(push_constant) uniform objectBuffer {  
    mat4 matrixObject;
    vec4 diffuse;
} object;
```

New values appended “in-band”: in the Command-Buffer

Efficient; but good for small amount of values
**VULKAN COMPONENTS**

- **Heap**
  - **Image**
  - **Image View**
    - **Buffer**
    - **Sampler**
  - **Framebuffer**
  - **Cmd.Buffer Pool**
- **Memory**
- **Device**
  - **Command-buffer**
    - **Barrier synchronization**
    - **Begin Render-Pass**
    - **Bind Graphics-pipeline**
    - **Set misc. dynamic states**
    - **Bind Vertex/Idx Buffer(s)**
    - **Update Buffer**
    - **Bind Descriptor-Set(s)**
    - **Draw...**
    - **Execute Commands**
    - **End Render-Pass**
- **Queue**
  - **Cmd.Buffer Pool**
  - **Begin Render-Pass**
  - **Bind Graphics-pipeline**
  - **Set misc. dynamic states**
  - **Bind Vertex/Idx Buffer(s)**
  - **Update Buffer**
  - **Bind Descriptor-Set(s)**
  - **Draw...**
  - **Execute Commands**
  - **End Render-Pass**
**IMAGES AND IMAGEVIEW**

**Images** represent all kind of ‘pixel-like’ arrays

- **Textures**: Color or Depth-Stencil
- **Render targets**: Color and Depth-Stencil
- Even Compute data

**ImageView** required to expose Images properly when specific format required

For Shaders

For Framebuffers
HOW DOES IT LOOK?

Simple texture creation

Way more complex than OpenGL!

- Load image
- Create an Image (1D/2D/3D/Cube...)
- Create an Image-View
- Aggregate layers/mipmap layers info (offsets, sizes) in a structure (VkBufferImageCopy)
- Aggregate layers & mipmap data to contiguous memory

- Create staging buffer + bind memory + copy data in it
- Use command-buffer to copy to the image: layers and mipmaps
  - Layout transition of image for copy
  - vkCmdCopyBufferToImage
  - Layout transition of image for use by shader
- Enqueue command buffer and execute
HOW DOES IT LOOK?

Simple texture creation
VULKAN COMPONENTS

Heap

Memory

Image

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

Cmd.Buffer Pool

Command-buffer

Barrier synchronization

Begin Render-Pass

Bind Graphics-pipeline

Set misc. dynamic states

Bind Vertex/Idx Buffer(s)

Update Buffer

Bind Descriptor-Set(s)

Draw...

Execute Commands

End Render-Pass

Device

Queue
Each DescriptorSet holds references to some resources.

Descriptor-Set-Layout defines how resources must be put together in a DescriptorSet.

Command buffers can then efficiently bind any or them.

They must match what shaders of each stage expect!

GLSL Code:

```glsl
layout(std140, set=0, binding=0) uniform A { ... };
layout(std140, set=0, binding=1) uniform B { ... };
layout(std140, set=1, binding=2) uniform C { ... };
layout(set=0, binding=3) uniform sampler2D tex;
... void main() { ... }
```
### VULKAN COMPONENTS

**Can use many if compatibles**

- **Framebuffer**
  - Simpler than OpenGL
  - “Bag” or “Repository” of resource views
  - No role defined for the resources

- **Render-Pass**
  - Really defines the role of Framebuffer resources
  - Can have more than 1 Sub-Pass
  - Each Sub-Passes defines which Framebuffer resource to use
  - invented for Tilers Arch
VULKAN COMPONENTS

Heap 1

Memory (Vid)

Memory (Sys)

Heap 2

Image View

Framebuffer

Render-Pass

Graphics pipeline

Buffer

Descriptor-Set

DescriptorSet Pool

2ndary Command-buffer

...
MEMORY ↔ VULKAN OBJECTS

Vulkan Objects referring to buffer(s) of data need binding to memory

- Vertex/Index Buffers; Uniform Buffers; Images/Textures...

Vulkan Device exposes various Memory Heaps - Example:

heap 0: size:12,288Mb (Video Memory of my K6000)
heap 1: size:17,911Mb (System Memory of my PC)

And various Memory Types from these Heaps. Example:

<table>
<thead>
<tr>
<th>Mem. Type</th>
<th>Heap</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (sys.mem)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0 (Video)</td>
<td>DEVICE_LOCAL</td>
</tr>
<tr>
<td>2</td>
<td>1 (sys.mem)</td>
<td>HOST_VISIBLE</td>
</tr>
<tr>
<td>3</td>
<td>1 (sys.mem)</td>
<td>HOST_VISIBLE</td>
</tr>
</tbody>
</table>

Tegra: Adds one more: HOST_VISIBLE “NON-Coherent”
RESOURCE MANAGEMENT
Allocation and Sub allocation

HEAP supporting A,B

Allocate memory type from heap

Allocation Type A

Image

Cube Image

Allocation Type B

Buffer

Query Vulkan Object about size, alignment & type requirements
Assign memory subregion to a resource (allows aliasing)

Create resource views on subranges of a buffer or image (array slices...)

HEAP supporting B
# HappyGPU

Memory Allocation

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Buffer</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Vertex</td>
<td>Uniform</td>
</tr>
</tbody>
</table>

Better...

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Buffer</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Vertex</td>
<td>Uniform</td>
</tr>
</tbody>
</table>

#HappyGPU

Bind same buffer with Offsets > 0
1 buffer can have many types of data
Vulkan uses SPIR-V passed directly to the driver

Can be compiled from GLSL Via glslang or LunarG’s glslangValidator; Google ShaderC

theoretically other languages could be compiled to Spir-V...

Libraries available to compile GLSL to Spir-V from the application

NVIDIA allows to compile GLSL directly

NVIDIA VK_NV_gls.shader: Vulkan reads GLSL directly

GLSL -> glslang -> SPIR

Some other language
SHADERS

Multiple entry points can be defined in a single Spir-V shader-module.

Prevents redundant code: shader module used by many Graphics-Pipelines.

Specialization Constants: early setup of constants for shaders in given Graphics-Pipeline.

Allows sharing snippets of code: easier to share common shader code.

Warning: Current GLSL → Spir-V compilers don’t support this feature, yet. But part of the API & Spir-V will happen soon.
VULKAN WINDOW SYSTEM INTEGRATION (WSI)

WSI manages the ownership of images via a swap chain

One image is presented while the other is rendered to

WSI is a Vulkan Extension

Swap Chain (images)

- Submit image to WSI
  The display owns the image

- Acquire image from WSI
  The application owns the image
NVIDIA OPENGL ↔ VULKAN INTEROP

Alternative to WSI: GL_NV_draw_vulkan_image

Create an OpenGL Context and all the usual things

Create Vulkan Device

Rendering Loop involves both OpenGL and Vulkan

Blit the Vulkan image to OpenGL backbuffer: glDrawVkImageNV

Extra care on synchronization (Semaphores)

Bonus: Mix OpenGL rendering (UI overlay…) with Vulkan

Allows smooth transition in projects
PRE-REQUISITES TO WORK WITH VULKAN


**Vulkan Loader** (+Source code)

**Tools:** Spir-V compiler for GLSL code and other libraries

**Layers:** intermediate code invoked by Vulkan API functions to help debug

**Vulkan Includes**

**Drivers:**

GeForce Experience


RECAP’ ON NVIDIA-SPECIFIC FEATURES

Compatible GPUs for Vulkan: Kepler and Higher; Shield Tablet; Shield Android TV

VK_NV_glsl_shader: GLSL can be directly sent to Vulkan

VK_NV_dedicated_allocation: more efficient memory usage

GL_NV_draw_vulkan_image can replace WSI

16 Queues. All available for all kind of use; 1 Queue for Copy-Engine only

3 frames (max) in flight with WSI

All Host memories are “Coherent” (except one for Tegra)

Layout transitions don’t exist in our HW (VK_IMAGE_LAYOUT_GENERAL)

Linear-Tiling only for 2D non-mipmapped textures... please avoid (bad performance)

Shaders never need re-compilation due to states in Graphics-pipeline
RECAP’ ON VULKAN PHILOSOPHY

Validate as much as possible up-front (DescriptorSets; Pipelines...)

The driver doesn’t waste time on figuring-out how to set things-up

Reuse existing patterns of Graphics-Pipelines: cached pipelines

Know your application: Taylor Vulkan design according to it

Know your memory usage: You are in charge of optimal sub-allocations

Explicit multi-threading for graphics: Application’s responsibility

Explicit Resource updates: Either through [non]Coherent buffers; or Queue-Based DMA transfers
FEW WORDS ON VKCPP PROJECT

C++11 to the rescue

• Open-Source Project of a C++11 overlay for Vulkan: became Khronos-offcial (!)

• Simplify Vulkan usage by
  • reducing risk of errors, i.e. type safety, automatic initialization of sType, ...
  • Reduce #lines of written code, i.e. constructors, initializer lists for arrays, ...
  • Add utility functions for common tasks (suballocators, resource tracking, ...)
VKCPP PROJECT
Two C++ based layers

Autogenerated ‘low-level’ layer using vulkan.xml

- Type safety
- Syntactic sugar
- Lightweight layer; Keeps you closer to the real Vulkan

Hand-coded ‘high level’ layer

- Reduce code complexity
- Exception safety, resource lifetime tracking, ...
- Closer dependency with VkCpp internal implementations
NATIVE VULKAN VS. VKCPP CODE

Native Vulkan: ~750 lines

vkCPP: ~200 lines
REFERENCES

Vulkan info from NVIDIA:

https://developer.nvidia.com/Vulkan

Samples + Source code in OpenGL and Vulkan:

https://github.com/nvpro-samples

Other:

https://gameworks.nvidia.com
https://developer.nvidia.com/designworks
http://vulkan.gpuinfo.org/listreports.php
THANK YOU

HTTPS://DEVELOPER.NVIDIA.COM/DESIGNWORKS

TLORACH@NVIDIA.COM