Introduction

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Acknowledgements

First of all, thanks to the inaugural class of 19 students who braved new, unrefined, and just-in-time course materials to take the first Vulkan class at Oregon State University – Winter Quarter, 2018. Thanks for your courage and patience!

Third, thanks to the Khronos Group for the great laminated Vulkan Quick Reference Cards! (Look at those happy faces in the photo holding them.)

Second, thanks to NVIDIA for all of their support!

Ali Alsalehy  
Natasha Anisimova  
Jianchang Bi  
Christopher Cooper  
Richard Cunard  
Braxton Cuneo  
Benjamin Fields  
Trevor Hammock  
Zach Lerew  
Victor Li  
Alan Neads  
Raja Petroff  
Bei Rong  
Lawrence Roy  
Lily Shellhammer  
Hannah Solorzano  
Jian Tang  
Glenn Upthagrove  
Logan Wingard
History of Shaders

2004: OpenGL 2.0 / GLSL 1.10 includes Vertex and Fragment Shaders

2008: OpenGL 3.0 / GLSL 1.30 adds features left out before

2010: OpenGL 3.3 / GLSL 3.30 adds Geometry Shaders

2010: OpenGL 4.0 / GLSL 4.00 adds Tessellation Shaders

2012: OpenGL 4.3 / GLSL 4.30 adds Compute Shaders

2017: OpenGL 4.6 / GLSL 4.60

There is lots more detail at:

History of Shaders

2014: Khronos starts Vulkan effort

2016: Vulkan 1.0

2016: Vulkan 1.1

2020: Vulkan 1.2

There is lots more detail at:
https://en.wikipedia.org/wiki/Vulkan_(API)
Everything You Need to Know is Right Here … Somewhere
Top Three Reasons that Prompted the Development of Vulkan

1. Performance
2. Performance
3. Performance

Vulkan is better at keeping the GPU busy than OpenGL is. OpenGL drivers need to do a lot of CPU work before handing work off to the GPU. Vulkan lets you get more power from the GPU card you already have.

This is especially important if you can hide the complexity of Vulkan from your customer base and just let them see the improved performance. Thus, Vulkan has had a lot of support and interest from game engine developers, 3rd party software vendors, etc.

As an aside, the Vulkan development effort was originally called “glNext”, which created the false impression that this was a replacement for OpenGL. It’s not.
Why is it so important to keep the GPU Busy?

### NVidia Titan V Specs vs. Titan Xp, 1080 Ti

<table>
<thead>
<tr>
<th>Feature</th>
<th>Titan V</th>
<th>Tesla V100</th>
<th>Tesla P100</th>
<th>GTX 1080 Ti</th>
<th>GTX 1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GV100</td>
<td>GV100</td>
<td>GP100 Cut-Down Pascal</td>
<td>GP102 Pascal</td>
<td>GP104-400 Pascal</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>21.1B</td>
<td>21.1B</td>
<td>15.3B</td>
<td>12B</td>
<td>7.2B</td>
</tr>
<tr>
<td>Fab Process</td>
<td>12nm FFN</td>
<td>12nm FFN</td>
<td>16nm FinFET</td>
<td>16nm FinFET</td>
<td>16nm FinFET</td>
</tr>
<tr>
<td>CUDA Cores / Tensor Cores</td>
<td>5120 / 640</td>
<td>5120 / 640</td>
<td>3584 / 0</td>
<td>3584 / 0</td>
<td>2560 / 0</td>
</tr>
<tr>
<td>TMUs</td>
<td>320</td>
<td>224</td>
<td>224</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>ROPs</td>
<td>?</td>
<td>96 (7)</td>
<td>88</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Core Clock</td>
<td>1200MHz</td>
<td>1328MHz</td>
<td>-</td>
<td>1607MHz</td>
<td></td>
</tr>
<tr>
<td>Boost Clock</td>
<td>1455MHz</td>
<td>1370MHz</td>
<td>1480MHz</td>
<td>1600MHz</td>
<td>1733MHz</td>
</tr>
<tr>
<td>FP32 TFLOPs</td>
<td>15TFLOPs</td>
<td>14TFLOPs</td>
<td>10.6TFLOPs</td>
<td>~11.4TFLOPs</td>
<td>9TFLOPs</td>
</tr>
<tr>
<td>Memory Type</td>
<td>HBM2</td>
<td>HBM2</td>
<td>HBM2</td>
<td>GDDR5X</td>
<td>GDDR5X</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>12GB</td>
<td>16GB</td>
<td>16GB</td>
<td>11GB</td>
<td>8GB</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>1.7Gbps HBM2</td>
<td>1.75Gbps HBM2</td>
<td>?</td>
<td>11Gbps</td>
<td>10Gbps GDDR5X</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>3072-bit</td>
<td>4096-bit</td>
<td>4096-bit</td>
<td>352-bit</td>
<td>256-bit</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>6Gbps/s</td>
<td>900GB/s</td>
<td>?</td>
<td>~484GB/s</td>
<td>320-32GB/s</td>
</tr>
<tr>
<td>Total Power Budget (“TDP”)</td>
<td>250W</td>
<td>250W</td>
<td>300W</td>
<td>250W</td>
<td>180W</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>1x 8-pin</td>
<td>1x 8-pin</td>
<td>?</td>
<td>1x 8-pin</td>
<td>1x 8-pin</td>
</tr>
<tr>
<td>Release Date</td>
<td>12/07/2017</td>
<td>4Q16-1Q17</td>
<td>TBD</td>
<td>5/27/2016</td>
<td></td>
</tr>
<tr>
<td>Release Price</td>
<td>$3000</td>
<td>$10000</td>
<td>-</td>
<td>$700</td>
<td>Reference: $700</td>
</tr>
</tbody>
</table>

The NVIDIA Titan V graphics card is not targeted at gamers, but rather at scientific and machine/deep learning applications. That does not, however, mean that the card is incapable of gaming, nor does it mean that we can't extrapolate future key performance metrics for Volta. The Titan V is a derivative of the earlier-released GV100 GPU, part of the Tesla accelerator card series. The key differentiator is that the Titan V ships at $3000, whereas the Tesla V100 was available as part of a $10,000 developer kit. The Tesla V100 still offers greater memory capacity by 4GB – 16GB HBM2 versus 12GB HBM2 – and has a wider memory interface, but other core features remain matched or nearly matched. Core count, for one, is 5120 CUDA cores on each GPU, with 640 Tensor cores (used for Tensorflow deep/machine learning workloads) on each GPU.
Who was the original Vulcan?

From WikiPedia:

“Vulcan is the god of fire including the fire of volcanoes, metalworking, and the forge in ancient Roman religion and myth. Vulcan is often depicted with a blacksmith's hammer. The Vulcanalia was the annual festival held August 23 in his honor. His Greek counterpart is Hephaestus, the god of fire and smithery. In Etruscan religion, he is identified with Sethlans. Vulcan belongs to the most ancient stage of Roman religion: Varro, the ancient Roman scholar and writer, citing the Annales Maximi, records that king Titus Tatius dedicated altars to a series of deities among which Vulcan is mentioned.”

Why Name it after the God of the Forge?
Who is the Khronos Group?

The Khronos Group, Inc. is a non-profit member-funded industry consortium, focused on the creation of open standard, royalty-free application programming interfaces (APIs) for authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. Khronos members may contribute to the development of Khronos API specifications, vote at various stages before public deployment, and accelerate delivery of their platforms and applications through early access to specification drafts and conformance tests.
Playing “Where’s Waldo” with Khronos Membership

PROMOTER MEMBERS

Over 100 members worldwide
Any company is welcome to join

OSU
Oregon State
University
Computer Graphics
Who’s Been Specifically Working on Vulkan?
Vulkan

• Originally derived from AMD’s *Mantle* API

• Also heavily influenced by Apple’s *Metal* API and Microsoft’s *DirectX 12*

• Goal: much less driver complexity and overhead than OpenGL has

• Goal: much less user hand-holding

• Goal: higher single-threaded performance than OpenGL can deliver

• Goal: able to do multithreaded graphics

• Goal: able to handle tiled rendering
Vulkan Differences from OpenGL

• More low-level information must be provided (by you!) in the application, rather than the driver

• Screen coordinate system is Y-down

• No “current state”, at least not one maintained by the driver

• All of the things that we have talked about being deprecated in OpenGL are really deprecated in Vulkan: built-in pipeline transformations, begin-end, fixed-function, etc.

• You must manage your own transformations.

• All transformation, color and texture functionality must be done in shaders.

• Shaders are pre-”half-compiled” outside of your application. The compilation process is then finished during the runtime pipeline-building process.
The Basic OpenGL Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

MC → WC → EC → EC

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

Framebuffer

Fragment Processing, Texturing, Per-fragment Lighting

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Shader-style

Vertex Shader

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

Fragment Shader

Fragment Processing, Texturing, Per-fragment Lighting

Framebuffer

Per-vertex in variables

Per-vertex out variables

Uniform Variables

Per-fragment in variables

Uniform Variables

gl_Vertex, gl_Normal, gl_Color

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix

gl_Position

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Vulkan-style

Per-vertex *in* variables

Uniform Variables

gl_Position, Per-vertex *out* variables

Rasterization

Per-fragment *in* variables

Uniform Variables

Output color(s)

Framebuffer

Fragment Shader

Vertex Shader
Moving part of the driver into the application

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets

Simpler drivers for low-overhead efficiency and cross vendor portability

Layered architecture so validation and debug layers can be unloaded when not needed

Run-time only has to ingest SPIR-V intermediate language

Unified API for mobile, desktop, console and embedded platforms

Khronos Group
Vulkan Highlights: Command Buffers

- Graphics commands are sent to command buffers
- E.g., `vkCmdDoSomething(cmdBuffer, ...);`
- You can have as many simultaneous Command Buffers as you want
- Buffers are flushed to Queues when the application wants them to be flushed
- Each command buffer can be filled from a different thread
Vulkan Highlights: Pipeline State Objects

• In OpenGL, your “pipeline state” is the combination of whatever your current graphics attributes are: color, transformations, textures, shaders, etc.

• Changing the state on-the-fly one item at-a-time is very expensive

• Vulkan forces you to set all your state variables at once into a “pipeline state object” (PSO) data structure and then invoke the entire PSO at once whenever you want to use that state combination

• Think of the pipeline state as being immutable.

• Potentially, you could have thousands of these pre-prepared pipeline state objects
**Vulkan: Creating a Pipeline**

- **VkGraphicsPipelineCreateInfo**
  - Shader stages
  - VertexInput State
  - InputAssembly State
  - Tessellation State
  - Viewport State
  - Rasterization State
  - MultiSample State
  - DepthStencil State
  - ColorBlend State
  - Dynamic State
  - Pipeline layout
  - RenderPass
  - basePipelineHandle
  - basePipelineIndex

- **VkPipelineShaderStageCreateInfo**
  - VkSpecializationInfo
  - which stage (VERTEX, etc.)

- **VkPipelineVertexInputStateCreateInfo**
  - VkVertexInputBindingDescription
  - binding
  - stride
  - inputRate
  - vertexInputAttributeDescription
  - location
  - binding
  - format
  - offset

- **VkViewportStateCreateInfo**
  - Viewport
  - x, y, w, h
  - minDepth
  - maxDepth
  - offset
  - extent

- **VkPipelineRasterizationStateCreateInfo**
  - CullMode
  - PolygonMode
  - FrontFace
  - LineWidth

- **VkPipelineDepthStencilStateCreateInfo**
  - DepthTestEnable
  - DepthWriteEnable
  - DepthCompareOp
  - StencilTestEnable
  - StencilOpStateFront
  - StencilOpStateBack

- **VkPipelineColorBlendStateCreateInfo**
  - BlendEnable
  - SrcColorBlendFactor
  - DstColorBlendFactor
  - ColorBlendOp
  - SrcAlphaBlendFactor
  - DstAlphaBlendFactor
  - AlphaBlendOp
  - ColorWriteMask

- **VkPipelineDynamicStateCreateInfo**
  - Array naming the states that can be set dynamically
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices );

This way of querying information is a recurring OpenCL and Vulkan pattern (get used to it):

result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );
Vulkan Code has a Distinct “Style” of Setting Information in *structs* and then Passing that Information as a pointer-to-the-struct

```c
VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = << buffer size in bytes >>
    vbci.usage = VK_USAGE_UNIFORM_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = nullptr;

VK_RESULT result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );

VkMemoryRequirements vmr;

result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr ); // fills vmr

VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.flags = 0;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = 0;

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &MatrixBufferMemoryHandle );

result = vkBindBufferMemory( LogicalDevice, Buffer, MatrixBufferMemoryHandle, 0 );
```
Vulkan Quick Reference Card

Steps in Creating Graphics using Vulkan

1. Create the Vulkan Instance
2. Setup the Debug Callbacks
3. Create the Surface
4. List the Physical Devices
5. Pick the right Physical Device
6. Create the Logical Device
7. Create the Uniform Variable Buffers
8. Create the Vertex Data Buffers
9. Create the texture sampler
10. Create the texture images
11. Create the Swap Chain
12. Create the Depth and Stencil Images
13. Create the RenderPass
14. Create the Framebuffer(s)
15. Create the Descriptor Set Pool
16. Create the Command Buffer Pool
17. Create the Command Buffer(s)
18. Read the shaders
19. Create the Descriptor Set Layouts
20. Create and populate the Descriptor Sets
21. Create the Graphics Pipeline(s)
22. Update-Render-Update-Render- …
Vulkan GPU Memory

• Your application allocates GPU memory for the objects it needs

• To write and read that GPU memory, you map that memory to the CPU address space

• Your application is responsible for making sure that what you put into that memory is actually in the right format, is the right size, has the right alignment, etc.
Vulkan Render Passes

- Drawing is done inside a render pass
- Each render pass contains what framebuffer attachments to use
- Each render pass is told what to do when it begins and ends
Vulkan Compute Shaders

- Compute pipelines are allowed, but they are treated as something special (just like OpenGL treats them)
- Compute passes are launched through dispatches
- Compute command buffers can be run asynchronously
Vulkan Synchronization

- Synchronization is the responsibility of the application.
- Events can be set, polled, and waited for (much like OpenCL).
- Vulkan itself does not ever lock – that’s your application’s job.
- Threads can concurrently read from the same object.
- Threads can concurrently write to different objects.
Vulkan Shaders

- GLSL is the same as before … almost

- For places it’s not, an implied
  ```
  #define VULKAN 100
  ```
  is automatically supplied by the compiler

- You pre-compile your shaders with an external compiler

- Your shaders get turned into an intermediate form known as SPIR-V (Standard Portable Intermediate Representation for Vulkan)

- SPIR-V gets turned into fully-compiled code at runtime

- The SPIR-V spec has been public for years –new shader languages are surely being developed

- OpenCL and OpenGL have adopted SPIR-V as well

**Advantages:**

1. Software vendors don’t need to ship their shader source
2. Software can launch faster because half of the compilation has already taken place
3. This guarantees a common front-end syntax
4. This allows for other language front-ends
Your Sample2019.zip File Contains This

The "19" refers to the version of Visual Studio, not the year of development.