Vulkan: the essentials
Tristan Lorach, March 17th 2016
Analogy On Graphic APIs
Analogy

Fixed-function OpenGL

Pre-assembled toy car

*fun out of the box, not much room for customization*
Analogy

Modern AZDO OpenGL with Programmable Shaders

LEGO Kit
you build it yourself,
comes with plenty of useful, pre-shaped pieces
Analogy

Vulkan

Pine Wood Derby Kit
you build it yourself to race from raw materials
power tools used to assemble, adult supervision highly recommended
Analogy

Different Valid Approaches

Fixed-function OpenGL  Modern AZDO OpenGL with Programmable Shaders  Vulkan
Beneficial Vulkan Scenarios

- **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
Beneficial Vulkan Scenarios

Start

Is your graphics work CPU bound? yes

Can your graphics creation be parallelized? yes

Tired with OpenGL (state-machine) or even D3D? yes

Want to learn new stuff? Kinda... (it's a Yes)

Kinda... Spend lots of time coding? No sleep? yes

You put a premium on avoiding hitches yes

You can manage your graphics resource allocations yes

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

OpenGL / D3D
Big Picture - Typical OpenGL Case

Application

OpenGL Commands

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

OpenGL Driver

Graphics pipeline States

Resources

Heap

Dependencies

Cmd bundles

GDC
Big Picture - Vulkan

Application memory

- GPU

- OpenGL Driver

- Minimal memory management

- Fewer translation, Validation checks And internal mgt

- Resources

- Pipeline States

- Rend Passes

- Descriptor Sets

- Dependencies

- Cmd-buffers / queues

- FBO resources (Textures / RB)

- Tr. Feedback buffer

- Uniform Block

- Texture Fetch

- Image Load/Store

- Atomic Counter

- Shader Storage

- Vertex Puller (IA)

- Vertex Shader

- TCS (Tessellation)

- Tessellator

- TES (Tessellation)

- Geometry Shader

- Transform Feedback

- Rasterization

- Fragment Shader

- Per-Fragment Ops

- Framebuffer

- Front-End (decoder)

- Push-Buffer (FIFO)
Vulkan Components

Heap
- Image View
- Image
- Memory

Memory
- Image View
- Image
- Buffer
- Sampler

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

Heap
- Image View
- Image
- Memory

Memory
- Image View
- Image
- Buffer
- Sampler

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
Vulkan Objects: Device

VkPhysicalDevice
- Capabilities
- Memory Management
- Queues
- Objects
  - Buffers
  - Images
  - Sync Primitives
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](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>alphaTo1X</td>
<td>true</td>
</tr>
<tr>
<td>depthClamp</td>
<td>true</td>
</tr>
<tr>
<td>depthBounds</td>
<td>true</td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
</tr>
<tr>
<td>dualSourceBlend</td>
<td>true</td>
</tr>
<tr>
<td>fillModeForSolid</td>
<td>true</td>
</tr>
<tr>
<td>fragmentStoresAndAtomicops</td>
<td>true</td>
</tr>
<tr>
<td>fullScreenBlend</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>inexactQueries</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>samplerAndropoly</td>
<td>true</td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderGpUDistance</td>
<td>true</td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>true</td>
</tr>
<tr>
<td>shaderInt16</td>
<td>true</td>
</tr>
<tr>
<td>shaderInt64</td>
<td>true</td>
</tr>
</tbody>
</table>

## Tegra X1 & K1

<table>
<thead>
<tr>
<th>Feature</th>
<th>NVIDIA NVIDIA Tegra X1</th>
<th>NVIDIA NVIDIA K1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td></td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>version</td>
<td>361.0.0 (1.0.2)</td>
<td>361.0.0 (1.0.2)</td>
<td></td>
</tr>
<tr>
<td>shader</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>alphaTo1X</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>depthBiasClamp</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>depthBounds</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>drawIndirectFirstInstance</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>dualSourceBlend</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>fillModeForSolid</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>fragmentStoresAndAtomicops</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>fullScreenBlend</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>geometryShader</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>imageCubeArray</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>independentBlend</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>inexactQueries</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>largePoints</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>logicOp</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>multiDrawIndirect</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>multiViewport</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>occlusionQueryPrecise</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>pipelineStatisticsQuery</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>robustBufferAccess</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>sampleRateShading</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>samplerAndropoly</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>shaderClipDistance</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>shaderGpUDistance</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>shaderInt16</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>shaderInt64</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>shaderImageGatherExtended</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>shaderInt16</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>shaderInt64</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>vertexPipelineStoresAndAtomicops</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>mipmapFilter</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
Queues

- 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: “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 16 Queues
    - Only one type of queue for all the types of work
Command-Buffers

- Vulkan Rendering ➔ Command-Buffers
- Almost 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 create Graphic Work from GPU (command-lists can): API calls to vkCmd...() between Begin & End
- Multi-threading friendly!
- Primary Cmd-Buffer can call many 2\textsuperscript{ndary} Cmd-Buffers
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
    
    ```
    layout(set=0, binding = 2) uniform MyBuffer {
      mat4 mW;
    }
    ```

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

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

• **events** and **barriers**
  • used to synchronize work **within a command buffer** or sequence of command buffers submitted to a single queue

• **fences**
  • used to synchronize work between the **device** and the **host**.
Command-Buffers and Multi-Threading

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

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

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

Thread 3
(Busy)
- Update Work
  - cmd. Buffer Pool
  - Create 2\textsuperscript{dary} Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

Thread 4
(Busy)
- Update Work
  - cmd. Buffer Pool
  - Create 2\textsuperscript{dary} Cmd Buffer
  - Feed Cmd Buffers
  - Give out Cmd Buffers

! Command Buffer Pool local to the thread!
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 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
Threads And Command Pools

- Threads can have more than 1 Command Pool
  - Ring-buffer: One Command-Pool per Frame
  - when that thread/frame is no longer in flight (Using Fences)
  - Faster to simply reset a pool
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

PipeLine cache

Descriptor-set Layout

Pipeline-layout

Graphics pipeline

Optional Dynamic States
- Viewport
- Scissor
- Blend const
- Stencil References
- Depth Bounds
- Depth Bias

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

Rasterizations State
- 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() { ... }
```
Vulkan Components
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...)
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
  - Shader Load/Store (imgLoadStore)
- **Image View** required to expose Images properly when specific format required
  - For Shaders
  - For Framebuffers
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!
Vulkan Components

- 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
- Heap
- Memory
- Image View
- Framebuffer
- Render-Pass
  - Graphics pipeline
  - Buffer
  - Descriptor-Set
- DescriptorSet Pool
  - 2ndary Command-buffer
    - ...
Vulkan Components

- **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)

- **Heap 2**
  - Memory (Sys)

- **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

- **Device**
  - **Queue**

- **Update Buffer**
  - **Set misc. dynamic states**
  - **Bind Vertex/Idx Buffer(s)**
  - **Bind Descriptor-Set(s)**
  - **Draw...**
  - **Execute Commands**
  - **End Render-Pass**
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
BufferView

Allocation Type B
Cube Image
BufferView

... 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...)

Buffer
Resource Management

Buffer  Buffer  Buffer  Buffer  Buffer  Buffer
Index  Vertex  Uniform  Index  Vertex  Uniform

Not. So. Good.

Better...

#HappyGPU
Bind same buffer with Offsets > 0
1 buffer can have many types of data
Shaders

• 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_glsl_shader`: Vulkan reads GLSL directly
Shaders

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

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

• 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.

---

Image of a diagram showing shader modules and their functions:

- **Module**
  - `rotate(...)`
  - `lighting()`

- Functions:
  - `vtx_main()`
  - `frag_wood()`
  - `frag_pastic()`
  - `Frag_metal()`

- Graphics pipelines:
  - **Graphics pipeline A**
    - `Vtx_main()`
    - `frag_pastic()`
  - **Graphics pipeline B**
    - `Vtx_main()`
    - `frag_metal()`
  - **Graphics pipeline C**
    - `Vtx_main()`
    - `frag_wood()`
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
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

• Lunar-G (http://lunarg.com/)
  • 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 (latest is 364.51 for a fix)
  • https://developer.nvidia.com/vulkan-driver
  • NVIDIA resources: https://developer.nvidia.com/Vulkan
Recap’ On NVIDIA-Specific Features

• Compatible GPUs for Vulkan: Kepler and Higher; Shield Tablet; Shield Android TV
• GLSL can be directly sent to Vulkan
• GL_NV_draw_vulkan_image can replace WSI
• 16 Queues. All available for any kind of use
• 2 frames 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
• Shaders never need re-compilation due to states in Graphics-pipeline
NSight for Vulkan
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
Thank you!

Feedback welcome: tlorach@nvidia.com

Vulkan info from NVIDIA:

• https://developer.nvidia.com/Vulkan

• https://developer.nvidia.com/vulkan-graphics-api-here

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