Vulkan on NVIDIA GPUs

Piers Daniell, Driver Software Engineer, OpenGL and Vulkan

Who am I?

Piers Daniell @piers_daniell

Driver Software Engineer - OpenGL, OpenGL ES, Vulkan

NVIDIA Khronos representative since 2010

- OpenGL, OpenGL ES and Vulkan
- Author of several extensions and core features

Technical lead for OpenGL driver updates 4.1 through 4.5

Technical lead for OpenGL ES 1.1 through ES 3.1+AEP on desktop

Technical lead for Vulkan driver

11+ years with NVIDIA

Agenda

Vulkan Primer

Vulkan on NVIDIA GPUs

Vulkan Primer

What is Vulkan?

What developers have been asking for

Reduce CPU overhead

Scale well with multiple threads

Precompiled shaders

Predictable – no hitching

Clean, modern and consistent API – no cruft

Native tiling and mobile GPU support

Why is Vulkan important?

The only cross-platform next-generation 3D API

Vulkan is the only cross-platform next generation API

- DX12 – Windows 10 only
- Metal – Apple only

Vulkan can run (almost) anywhere

- Windows - XP, Vista, 7, 8, 8.1 and 10
- Linux
- SteamOS
- Android (as determined by supplier)

Who's behind Vulkan?

Hardware vendors

SONY

NVIDIA

ARM

QUALCOMM

MEDIATEK

AMD

intel

BROADCOM

Imagination

VIVANTE

SAMSUNG

* not a complete list!

Who’s behind Vulkan?

Software vendors

* not a complete list!

Vulkan for all GPUs

Low-power mobile through high-performance desktop

Vulkan is one API for all GPUs

Vulkan supports optional fine-grained features and extensions

- Platforms may define feature sets of their choosing

Supports multiple vendors and hardware

- From ES 3.1 level hardware to GL 4.5 and beyond
- Tile-based [deferred] hardware - Mobile
- Feed-forward rasterizing hadware - Desktop

Vulkan release

When can we get it?

Khronos' goal by the end of 2015

This discussion on the API is high-level

Details may change before release!

Vulkan conformance

Ensuring consistent behavior across all implementations

Conformance tests under development by Khronos

Includes large contributions from several member companies

Goal to release full conformance suite with Vulkan 1.0 release

Implementation must pass conformance to claim Vulkan support

Hello Triangle

Quick tour of the API

Launch driver and create display

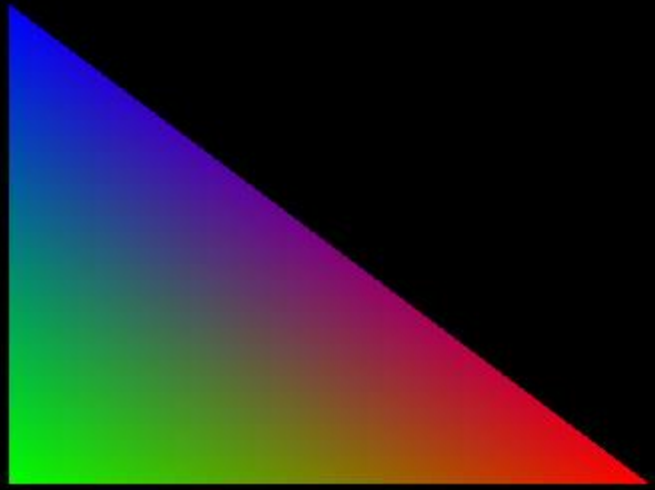
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



Vulkan Loader

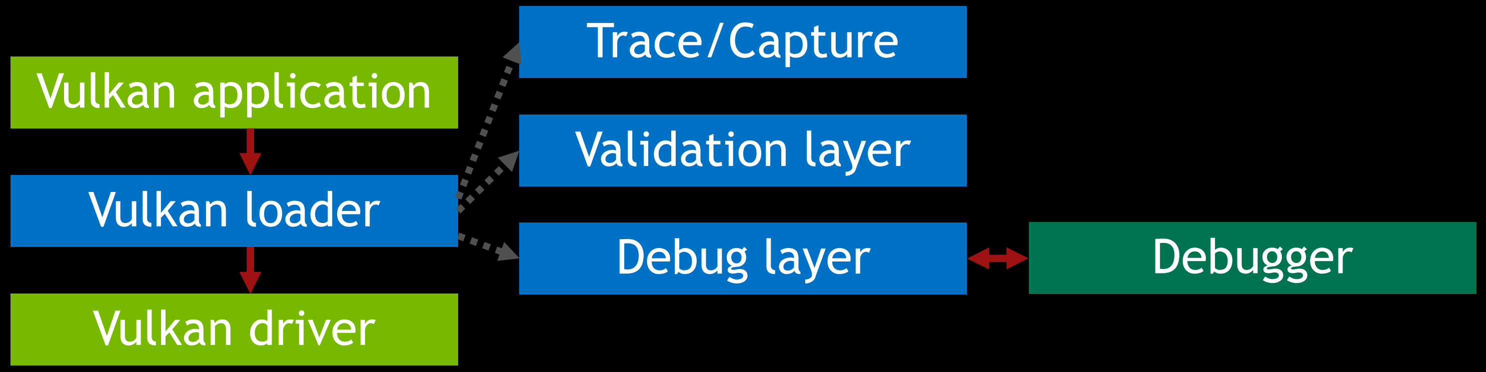
Part of the Vulkan ecosystem

Khronos provided open-source loader

Finds driver and dispatches API calls

Supports injectable layers

Validation, debug, tracing, capture, etc.



LunarG GLAVE debugger

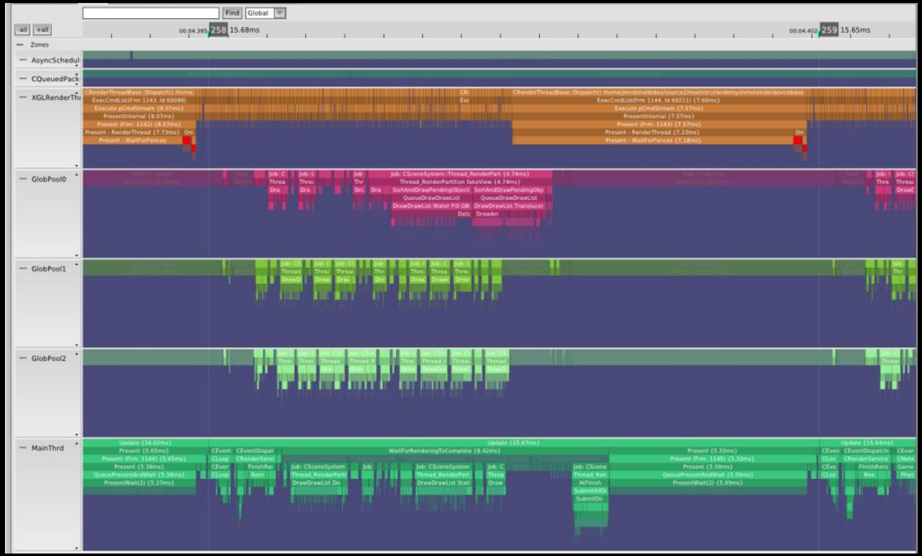
And other tools

LunarG and Valve working to create open-source Vulkan tools

Vulkan will ship with an SDK

More info and a video of GLAVE in action:

http://lunarg.com/Vulkan/



Vulkan Window System Integration

WSI for short

Khronos defined Vulkan extensions

Creates presentation surfaces for window or display

Acquires presentable images

Application renders to presentable image and enqueues the presentation

Supported across wide variety of windowing systems

- Wayland, X, Windows, etc.

Hello Triangle

Quick tour of the API

Launch driver and create display

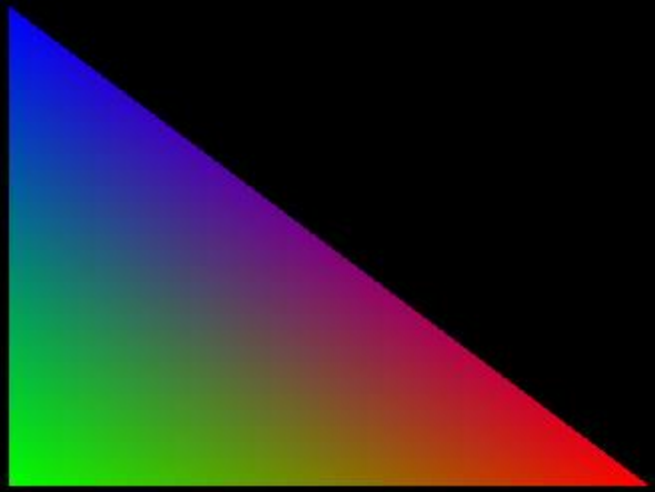
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



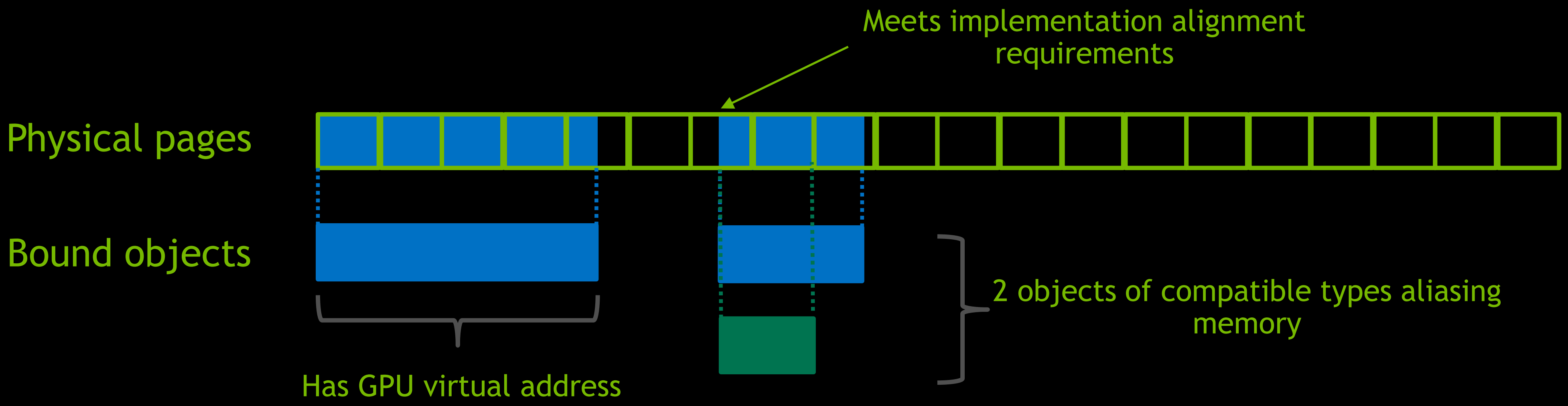
Low-level memory control

Console-like access to memory

Vulkan exposes several physical memory pools – device memory, host visible, etc.

Application binds buffer and image virtual memory to physical memory

Application is responsible for sub-allocation



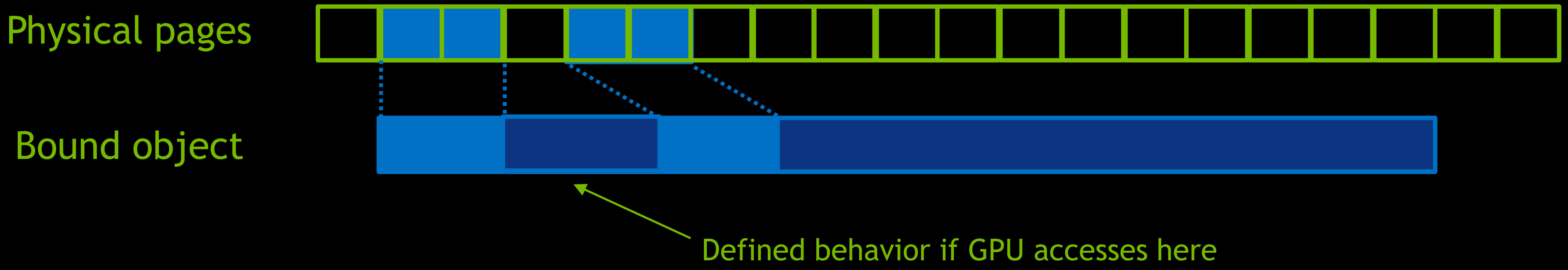
Sparse memory

More control over memory usage

Not all virtual memory has to be backed

Several feature levels of sparse memory supported

ARB_sparse_texture, EXT_sparse_texture2, etc.



Resource management

Populating buffers and images

Vulkan allows some resources to live in CPU-visible memory

Some resources can only live in high-bandwidth device-only memory

- Like specially formatted images for optimal access

Data must be copied between buffers

Copy can take place in 3D queue or DMA/copy queue

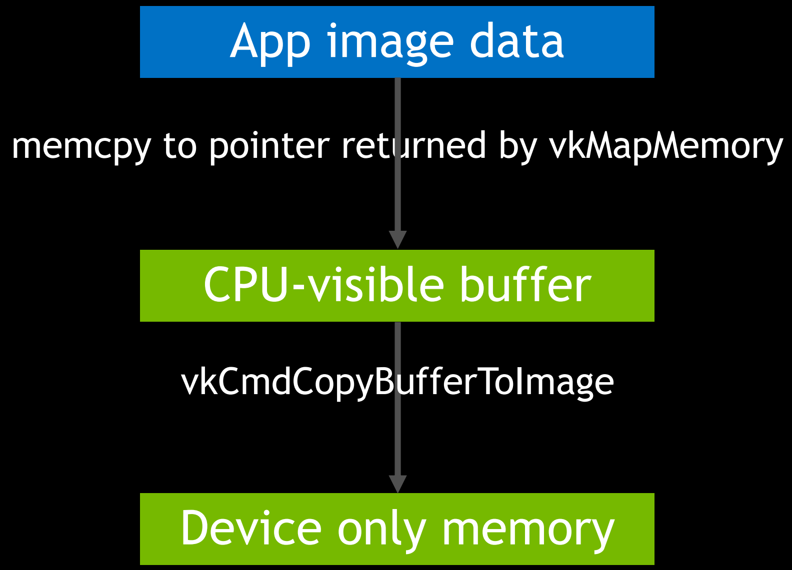
Copies can be done asynchronously with other operations

- Streaming resources without hitching

Populating vidmem

Using staging buffers

- Allocate CPU-visible staging buffers
    - These can be reused
- Get a pointer with vkMapMemory
    - Memory can remain mapped while in use
- Copy from staging buffer to device memory
    - Copy command is queued and runs async
- Use vkFence for application to know when xfer is done
- Use vkSemaphore for dependencies between command buffers



Descriptor sets

Binding resources to shaders

Shader resources declared with binding slot number

```
layout(set = 1, binding = 3) uniform image2D myImage;
layout(set = 1, binding = 4) uniform sampler mySampler;
```

Descriptor sets allocated from a descriptor pool

Descriptor sets updated at any time when not in use

- Binds buffer, image and sampler resources to slots

Descriptor set bound to command buffer for use

- Activates the descriptor set for use by the next draw

Multiple descriptor sets

Partitioning resources by frequency of update

Application can modify just the set of resources that are changing

Keep amount of resource binding changes as small as possible

Shader code

```
layout(set=0,binding=0) uniform { ... } sceneData;
layout(set=1,binding=0) uniform { ... } modelData;
layout(set=2,binding=0) uniform { ... } drawData;

void main() { }
```

Application code

```
foreach (scene) {
  vkCmdBindDescriptorSet(0, 3, {sceneResources,modelResources,drawResources});
  foreach (model) {
    vkCmdBindDescriptorSet(1, 2, {modelResources,drawResources});
    foreach (draw) {
      vkCmdBindDescriptorSet(2, 1, {drawResources});
      vkDraw();
    }
  }
}
```

Hello Triangle

Quick tour of the API

Launch driver and create display

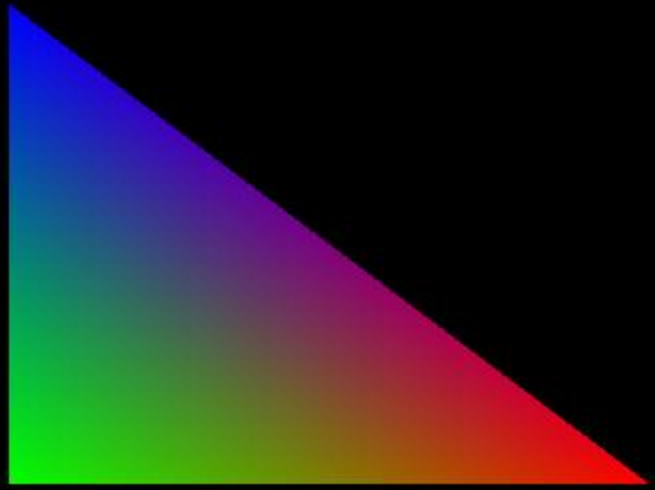
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



SPIR-V

Intermediate shader representation

Portable binary representation of shaders and compute kernels

Can support a wide variety of high-level languages including GLSL

Provides consistent front-end and semantics

Offline compile can save some runtime compile steps

The only shader representation accepted by Vulkan

- High-level shaders must be compiled to SPIR-V

SPIR-V

For your content pipeline

Khronos supported open-source GLSL->SPIR-V compiler - glslang

ISVs can easily incorporate into their content pipeline

- And use their own high-level language

SPIR-V provisional specs already published

Start preparing your content pipeline today!

Vulkan shader object

Compiling the SPIR-V

SPIR-V passed into the driver

Driver can compile everything except things that depend on pipeline state

Shader object can contain an uber-shader with multiple entry points

- Specific entry point used for pipeline instance

Reuse shader object with multiple pipeline state objects

Pipeline state object

Say goodbye to draw-time validation

Represents all static state for entire 3D pipeline

- Shaders, vertex input, rasterization, color blend, depth stencil, etc.

Created outside of the performance critical paths

Complete set of state for validation and final GPU shader instructions

- All state-based compilation done here – not at draw time

Can be cached for reuse

- Even across application instantiations

Pipeline cache

Reusing previous work

Application can allocate and manage pipeline cache objects

Pipeline cache objects used with pipeline creation

- If the pipeline state already exists in the cache it is reused

Application can save cache to disk for reuse on next run

Using the Vulkan device UUID – can even stash in the cloud

Pipeline layout

Using compatible pipelines

Pipeline layout defines what kind of resource is in each binding slot

- Images, Samplers, Buffers (UBO, SSBO)

Different pipeline state objects can use the same layout

- Which means shaders need to use the same layout

Changing between compatible pipelines avoids having to rebind all descriptions

- Or use lots of different descriptor sets

Dynamic state

State that can change easily

Dynamic state changes don’t affect the pipeline state

- Does not cause shader recompilation

Viewport, scissor, color blend constants, polygon offset, stencil masks and refs

All other state has the potential to cause a shader recompile on some hardware

- So it belongs in the pipeline state object with the shaders

Hello Triangle

Quick tour of the API

Launch driver and create display

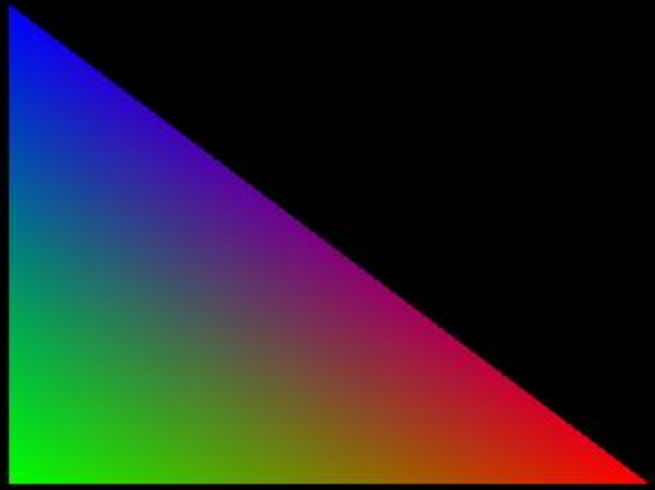
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



Renderpass

Units of work for tiler-friendly rendering

- Application defines how framebuffer cache is populated at start
  - Loaded from real framebuffer, cleared or ignored
- Application defines how framebuffer cache is flushed at the end
  - Stored back to real framebuffer, multi-sample resolved or discarded
- Application can chain multiple render-passes together
  - Execute all passes and eliminate framebuffer bandwidth between each pass
  - Example: gbuffer creation, light accumulation, final shading and post-process all without framebuffer traffic between steps

Command buffers and pools

A place for the GPU commands

A command buffer is an opaque container of GPU commands

Command buffers are submitted to a queue for the GPU to schedule execution

Commands are adding when the command buffer is recorded

Memory for the command buffer is allocated from the command pool

Multiple command buffers can allocate from a command pool

Commands and command buffers

Building a command buffer

Start a render pass

Bind all the resources

- Descriptor set(s)
- Vertex and Index buffers
- Pipeline state

Modify dynamic state

Draw

End render pass

Repeat: change any state and draw

Command buffer performance

Command buffer recording needs to scale well

Recording command buffers is the most performance critical part

- But we have no idea how big command buffer will end up

Can record multiple command buffers simultaneously from multiple threads

Command pools ensure there is no lock contention

- True parallelism provides multi-core scalability

Command buffer can be reused, re-recorded or recycled after use

- Reuse previous allocations by the command pool

Multi-threading

Maximizing parallel multi-CPU execution

Vulkan is designed so all performance critical functions don’t take locks

- Application is responsible for avoiding hazards

Use different command buffer pools to allow multi-CPU command buffer recording

Use different descriptor pools to allow multi-CPU descriptor set allocations

Most resource creation functions take locks

- But these are not on the performance path

Compute

For all your general-purpose computational needs

Uses a special compute pipeline

Uses the same descriptor set mechanism as 3D

- And has access to all the same resources

Can be dispatched interleaved with render-passes

- Or to own queue to execute in parallel

Resource hazards

Application managed

Resource use from different parts of the GPU may have read/write dependencies

- For example, will writes to framebuffer be seen later by image sampling

Application uses explicit barriers to resolve dependencies

- GPU may flush/invalidate caches so latest data is written/seen

Platform needs vary substantially

- Application expresses all resource dependencies for full cross-platform support

Application also manages resource lifetime

- Can’t destroy a resource until all uses of it have completed

Avoiding hazards

An example - sampling from modified image

Update an image with shader imageStore() calls

```
vkBindPipeline(cmd, pipelineUsesImageStore);
vkDraw(cmd);
```

Flush imageStore() cache and invalidate image sampling cache

```
vkPipelineBarrier(cmd, image, SHADER_WRITE, SHADER_READ);
```

Can now sample from the updated image

```
vkBindPipeline(cmd, pipelineSamplesFromImage);
vkDraw(cmd);
```

Hello Triangle

Quick tour of the API

Launch driver and create display

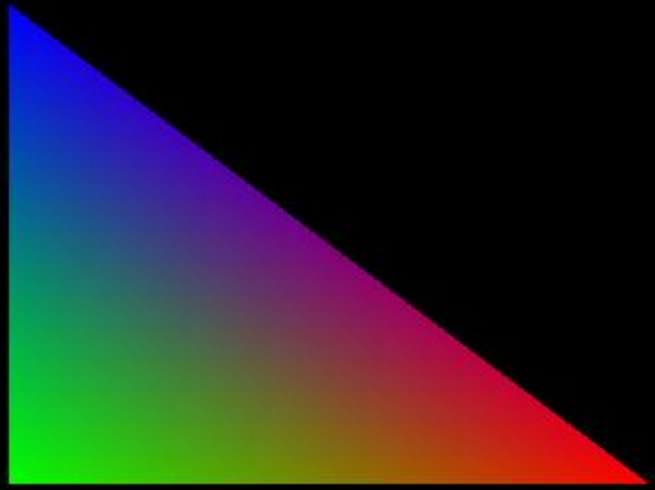
Set up resources

Set up the 3D pipe

- Shaders
- State

Record commands

Submit commands



Queue submission

Scheduling the commands in the GPU

Implementation can expose multiple queues

- 3D, compute, DMA/copy or universal

Queue submission should be cheap

Queue execution is asynchronous

App uses vkFence to know when work is done

App can use vkSemaphore to synchronize dependencies between command buffers

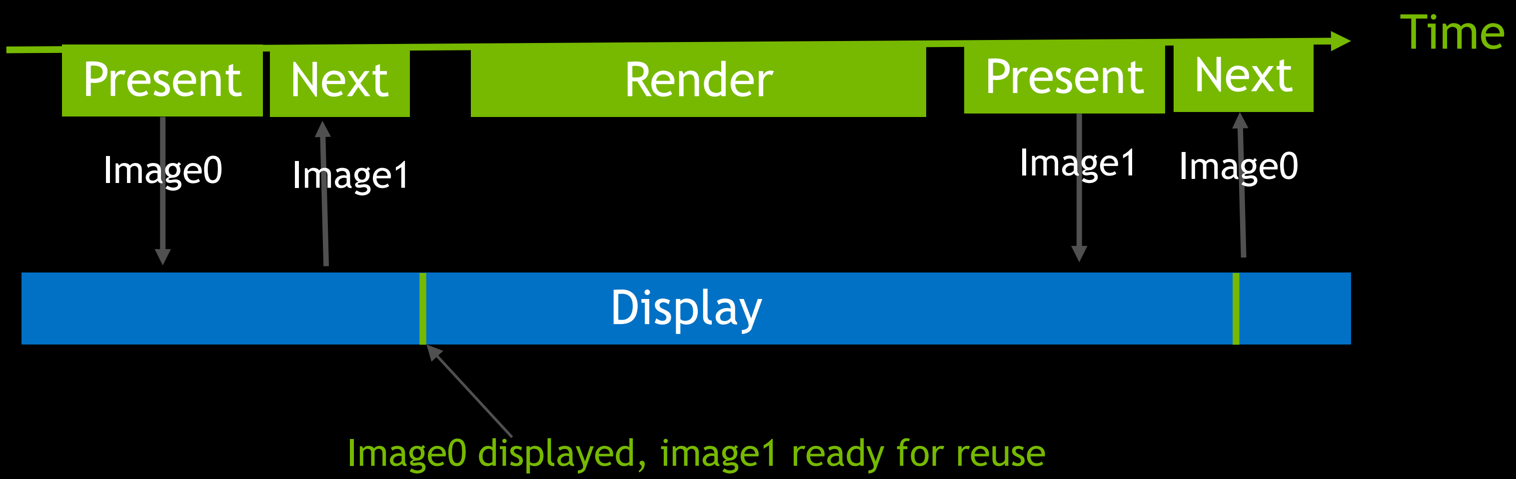
Presentation

Using the WSI extension

The final presentable image is queued for presentation

Presentation happens asynchronously

After present is queued application picks up next available image to render to



GFXBench 5.0

Early alpha content for Vulkan

Developed by Kishonti – maker of GFXBench

Entirely new engine aimed at benchmarking low-level graphics APIs

- Vulkan, DX12, Metal

Concept is a night outdoor scene with aliens

Still in alpha for Vulkan, but shows the most important concepts

Demo: GFXbench 5 alpha

Running on Windows 10

Vulkan on NVIDIA GPUs

Why is it important to NVIDIA?

It's open

API is designed to be extensible

- We can easily expose new GPU features

No single vendor or platform owner controls the API

Scales from low-power mobile to high-performance desktop

Can be used on any platform

It's fast!

What about OpenGL?

OpenGL is also very important to NVIDIA

OpenGL and OpenGL ES will remain vital

- Together have largest 3D API market share
- Applications – games, design, medical, science, education, film-production, etc.

OpenGL improvements since last year

- Maxwell extensions (15 of them!) – EXT_post_depth_coverage, EXT_raster_multisample, EXT_sparse_texture2, EXT_texture_filter_minmax, NV_conservative_raster, NV_fill_rectangle, NV_fragment_shader_interlock, etc.
- NV_command_list, OpenGL ES Android Extension Pack, bindless UBO, etc.

Even more improvements? Come to the Khronos BOF to find out!

OpenGL vs Vulkan

Solving 3D in different ways

- OpenGL higher-level API, easier to teach and prototype in
  - Many things handled automatically
- OpenGL can be used efficiently and obtain great single-threaded performance
  - Use multi-draw, bindless, persistently mapped buffers, PBO, etc.
- Vulkan’s ace is its ability to scale across multiple CPU threads
  - Can be used with almost no lock contention on the performance critical path
  - OpenGL does not have this (yet?)

Vulkan on NVIDIA GPUs

Fully featured

Vulkan is one API for all GPUs

Vulkan API supports optional features and extensions

Supports multiple vendors and hardware

- From ES 3.1 level hardware to GL 4.5 and beyond

NVIDIA implementation fully featured

- From Tegra K1 through GeForce GTX TITAN X

Write once run everywhere

Vulkan GPU support

ARCHITECTURE	GPUS
Fermi	GeForce 400 and 500 series Quadro x00 and x000 series
Kepler	GeForce 600 and 700 series Quadro Kxxx series Tegra K1
Maxwell	GeForce 900 series and TITAN X Quadro Mxxx series Tegra X1

Vulkan feature support

FEATURE	FERMI	KEPLER	MAXWELL
OpenGL ES 3.1 level features	Yes	Yes	Yes
OpenGL 4.5 level features	Yes	Yes	Yes
Sparse memory	Partial	Partial	Yes
ETC2, ASTC texture compression	No	Tegra	Tegra

Vulkan OS support

Everywhere we can

Windows XP, Vista, 7, 8, 8.1 and 10

Linux

SteamOS

Android – SHIELD Tablet and SHIELD Android TV

NVIDIA implementation walkthrough

Using GameWorks cadscene sample

GL version is open source

Vulkan version will be made available after spec release

CPU bound under OpenGL with large models

GPU bound on Vulkan!

The NVIDIA Vulkan driver

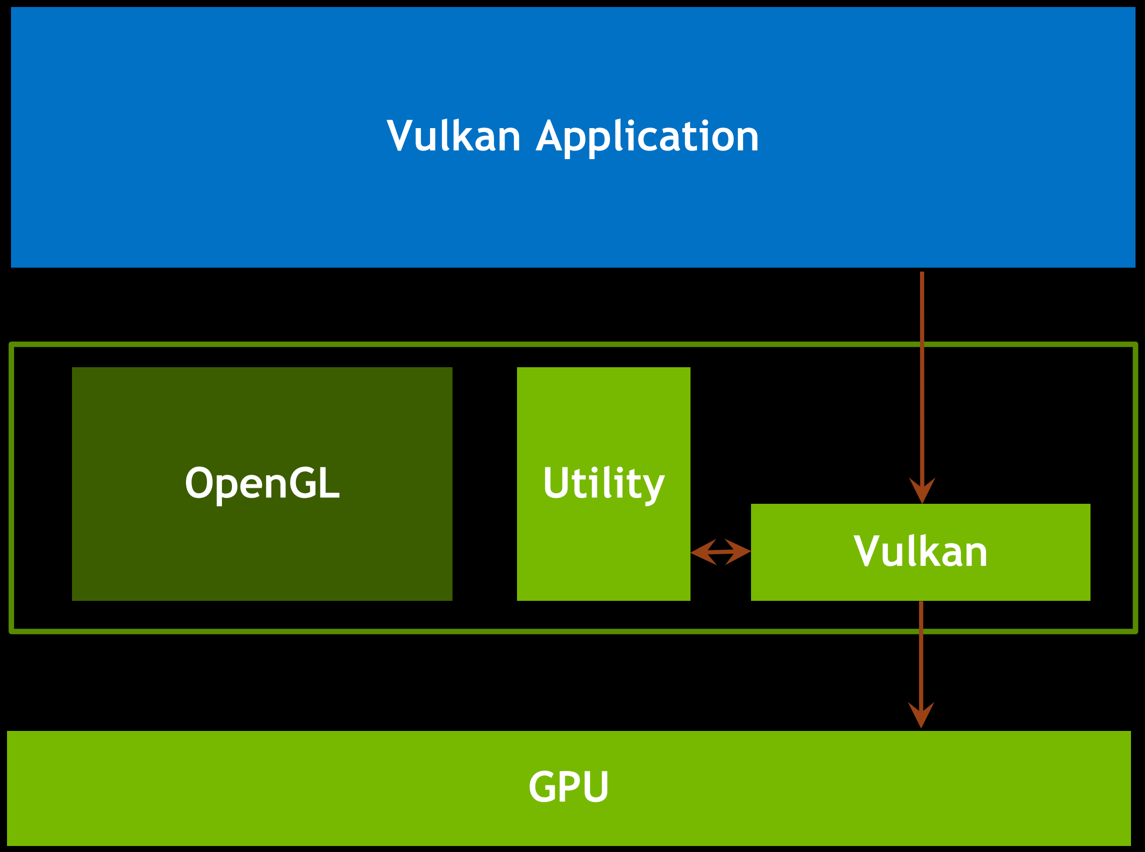
Hosted by the OpenGL driver

OpenGL and Vulkan share driver

OpenGL portion dormant

Performance critical path direct to GPU

Utility for resource and GPU management



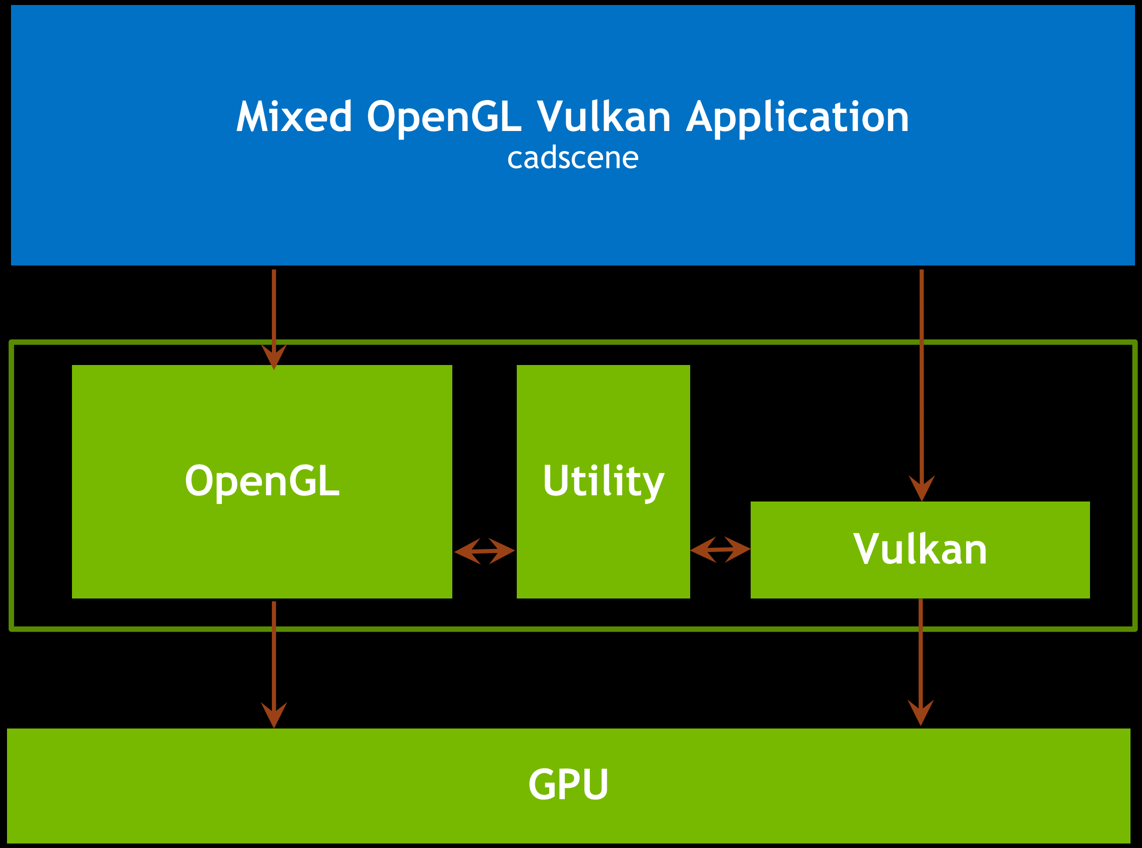
Vulkan and OpenGL

Living happily together

OpenGL and Vulkan paths to hardware remain separate

Can share resources

Performance optimal



Benefits of mixed driver

Efficiency for all

Ease transition to Vulkan

Allows applications to incrementally add Vulkan where it matters most

If you can get OpenGL, you can get Vulkan

Leveraged driver development

From OpenGL to Vulkan

Porting your existing code

Take incremental steps – using AZDO (Aproaching zero-overhead driver)

- http://www.slideshare.net/CassEveritt/approaching-zero-driver-overhead
- Persistent buffers, multi-draw indirect, bindless resources, etc.

Start using NV_command_list

- See "Best of GTC" talk on NV_command_list Monday 2pm by Tristan Lorach

Port performance-critical parts to Vulkan

- Can leave other stuff in OpenGL

Vulkan goals

How do we meet these goals?

Reduce CPU overhead

Scale well with multiple threads

Predictable – no hitching

Mobile GPU support

Demo: Vulkan cadscene

CPU overhead, multi-CPU scaling, pipeline changes

cadscene on Shield

Using the GameWorks cross-platform SDK

Same framework used for NVIDIA GameWorks samples

    https://github.com/NVIDIAGameWorks

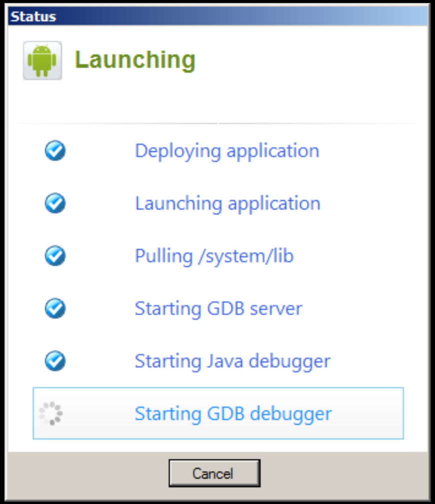
Supports cross-platform development

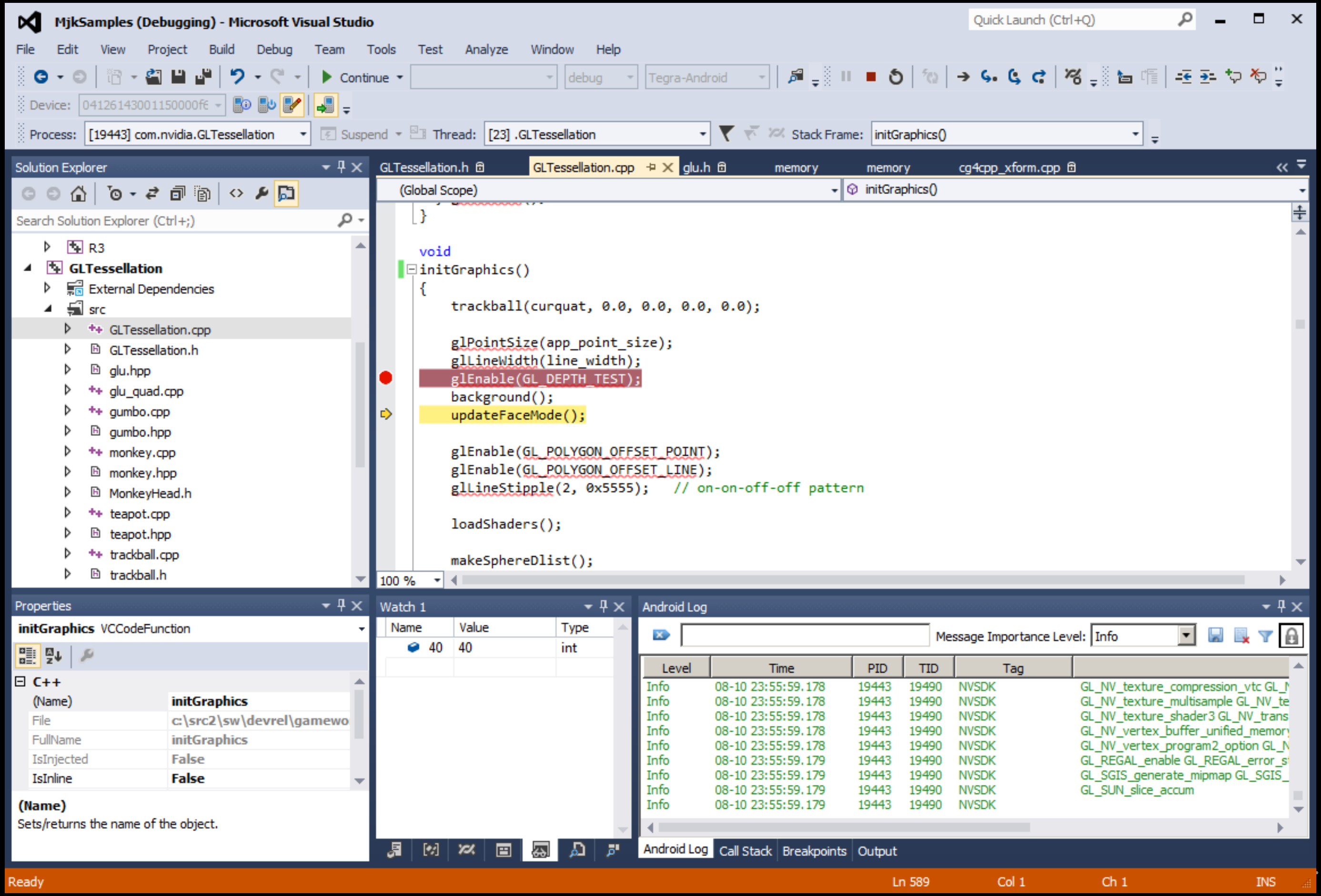
    Code for Windows, Linux and Android

GameWorks framework

Build, deploy and debug Android code right from Visual Studio

Coming for Vulkan…





Demo: Vulkan cadscene on Shield

Interactive high-polygon count CAD models

Vulkan driver

And how do I get one?

- Before Vulkan spec release
  - Become a Khronos member
  - Sign an NDA
- After Vulkan spec release (later this year!)
  - Download from nvidia.com

More Vulkan at SIGGRAPH

Don't miss a thing

Course: Moving Mobile Graphics

Sunday 2pm – 5:15pm

Course: An Overview of Next-Generation Graphics APIs

Tuesday 9am – 12:15pm

Khronos Birds of a Feather

Wednesday 5:30pm – 7:30pm

Party! 7:30pm – 10pm

Thank you!

Get your **free** Khronos Vulkan t-shirts!

Questions?

Piers Daniell, Driver Software Engineer, OpenGL and Vulkan