Vulkan SC

Mukund Keshava, System Software Engineer

Sruthik P, System Software Engineer

Agenda

- Introduction to Vulkan
- Why Vulkan SC
- Basic Philosophy
- Vulkan SC vs Vulkan
- Vulkan SC Workflow
- Code examples

Introduction to Vulkan

Graphics APIs before Vulkan – Fixed Function Era

1983

SGI releases IRIS GL

One of the first proprietary Graphics API

1993

SGI releases OpenGL

- Open-source variant of IRIS GL.
- OpenGL API designed around configuring a monolithic state machine.
- Low-level GPU hardware mechanisms abstracted away from API.
- Large driver overhead

1996

Microsoft releases DirectX 2

Lower-level Graphics API targeting Windows OS

- Different approach compared to OpenGL
- Exposed two types of APIs: Immediate and Retained mode
- Uses concept of "execute buffers"

1997 - 2000

DirectX 3 - 7

Execute buffers not preferred by developers and IHVs.

- Later versions of DirectX move to OpenGL style Draw APIs and drop execute buffers.
- Retained Mode API is dropped in favor of Immediate Mode.

Graphics APIs before Vulkan – Programmable Era

2000

DirectX 8

One of the first Graphics APIs to support "programmable" Vertex and Fragment Shaders

2002

HLSL with DirectX 9

- One of the first High Level Shading languages. GPUs start becoming more programmable.

2004

GLSL with OpenGL 2.0

- Open-source Shading language

2005 - 2012

DirectX 10, 11 and GL 3,4

GPUs start becoming more programmable.

- More shader stages introduced (Tessellation, compute etc.)
- APIs are still relatively high level compared to consoles.
- GPGPU

Graphics APIs before Vulkan – Programmable Era (Low Level APIs)

2016

Khronos releases Vulkan 1.0

- Open-source, platform agnostic low level Graphics API.

2015

DirectX12

- Microsoft's low level Graphics API.
- Supported only on Windows OS.

Introduction

Vulkan

- New API from Khronos, introduced in 2016.
- Low level
- Cross platform – Desktop & Mobile
- Deterministic
- Low driver overhead
- Multi-threading
- Ray Tracing
- Current version: 1.3

Introduction

Comparison with OpenGL

OpenGL		Vulkan	
Implicit	Large driver overhead	Explicit	Lesser driver overhead
Internal memory management	Object model prevents efficient multi-threading	Memory manged by app	Efficient multi-threaded support
Fewer lines of code	WSI not part of specification	More lines of code	WSI part of specification

Introduction

Concepts

Pipeline	• Various pipelines supported – Graphics, Compute, RT. • Shader modules to be provided. SPIR-V
Memory	• All allocation done by application. Images, buffers need to be allocated and managed by app.
FrameBuffer	• Buffers into which the device renders into
RenderPass	• Render Description

Introduction

Concepts

Command Buffer	• Memory for recording device commands. Also needs to be allocated by the application.
Synchronization	• Explicitly done by the application via various synchronization primitives.
Swapchain	• Various backends present. Number of images specified by application.
Descriptors	• Describe Shader Resources

Introduction

Vulkan Design

Application Flow

Initial setup

Instance, Object creation

Create/Allocate objects

Buffers (vertex, index etc.), Images, Command buffers, Sync objects

Record commands

Command buffer recording

Submit commands

Submit command buffers to queue

Introduction

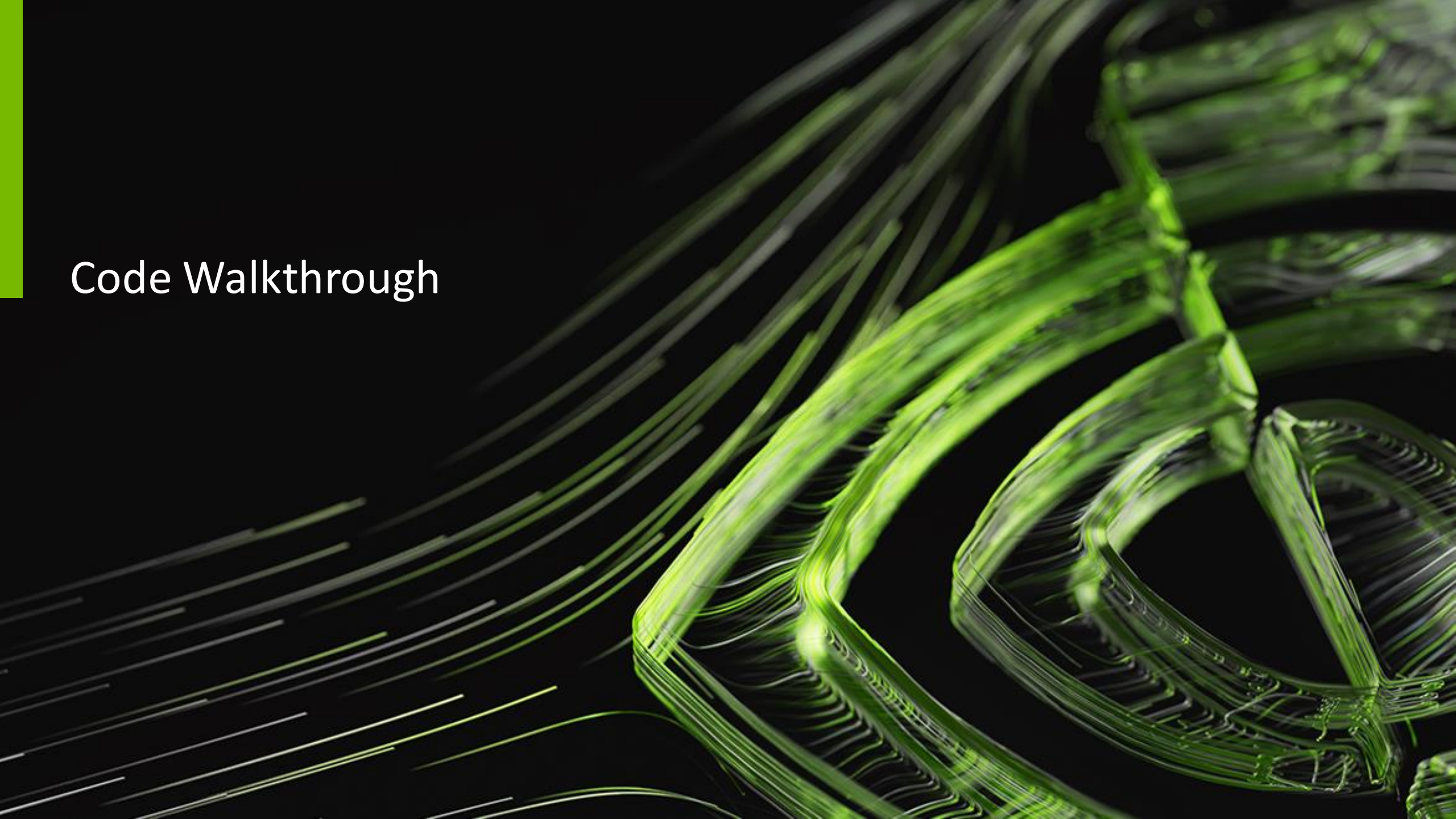
Basics

- Vulkan uses structures to feed information to the driver.
- There are “info” structures that need to be filled. Example:

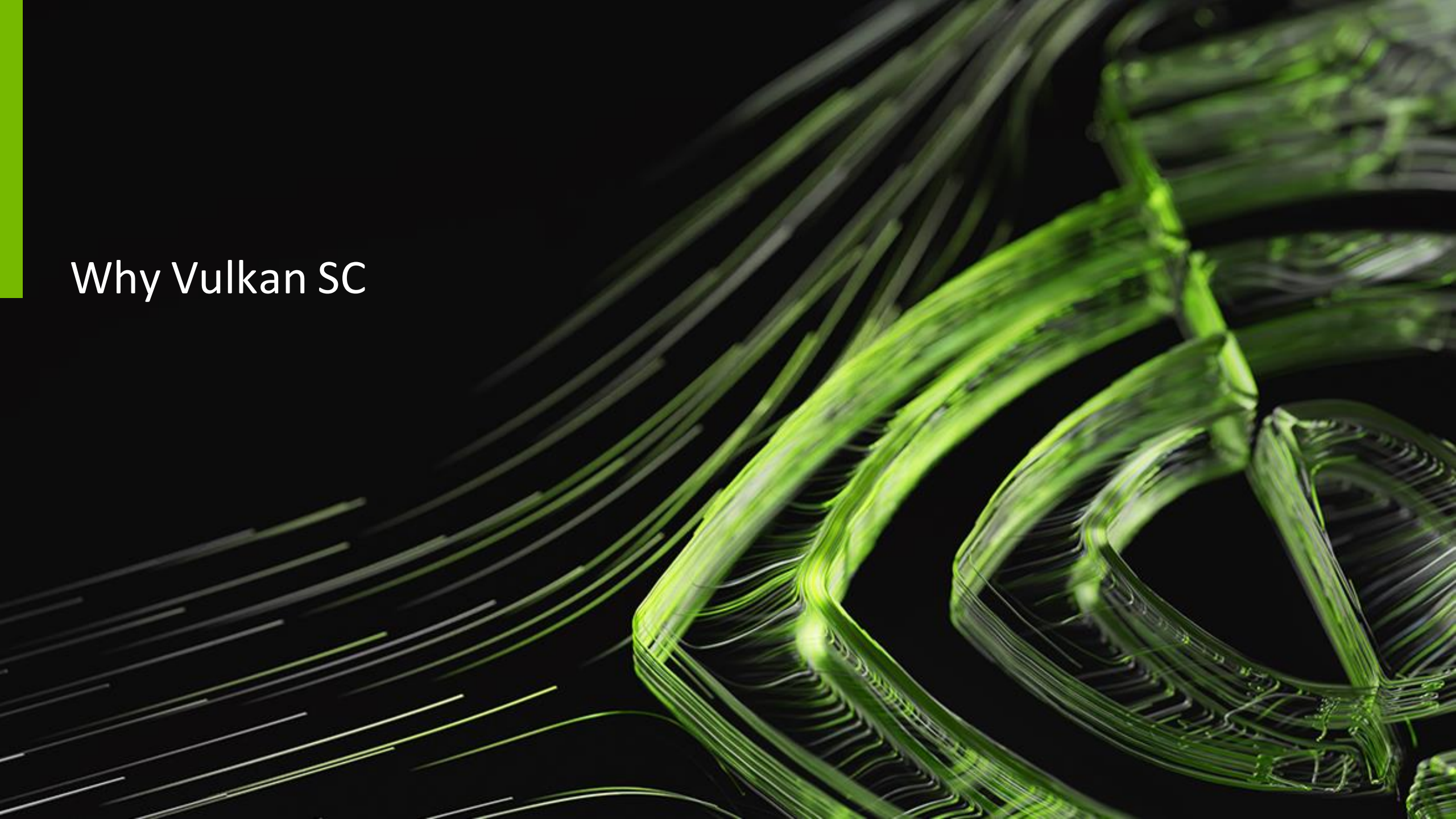
```
VkMemoryAllocateInfo allocInfo{};
allocInfo.sType           = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize  = size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory);
```

Code Walkthrough

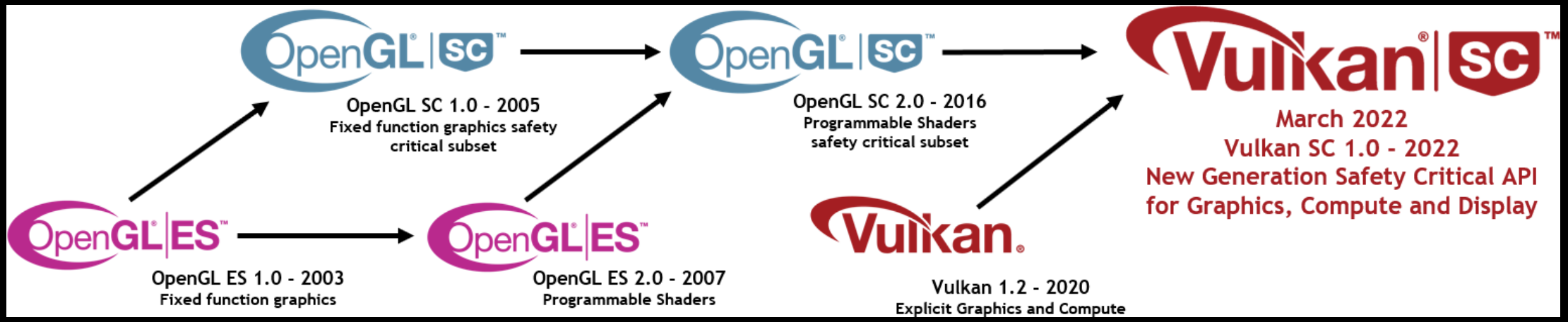


Why Vulkan SC



Why Vulkan SC?

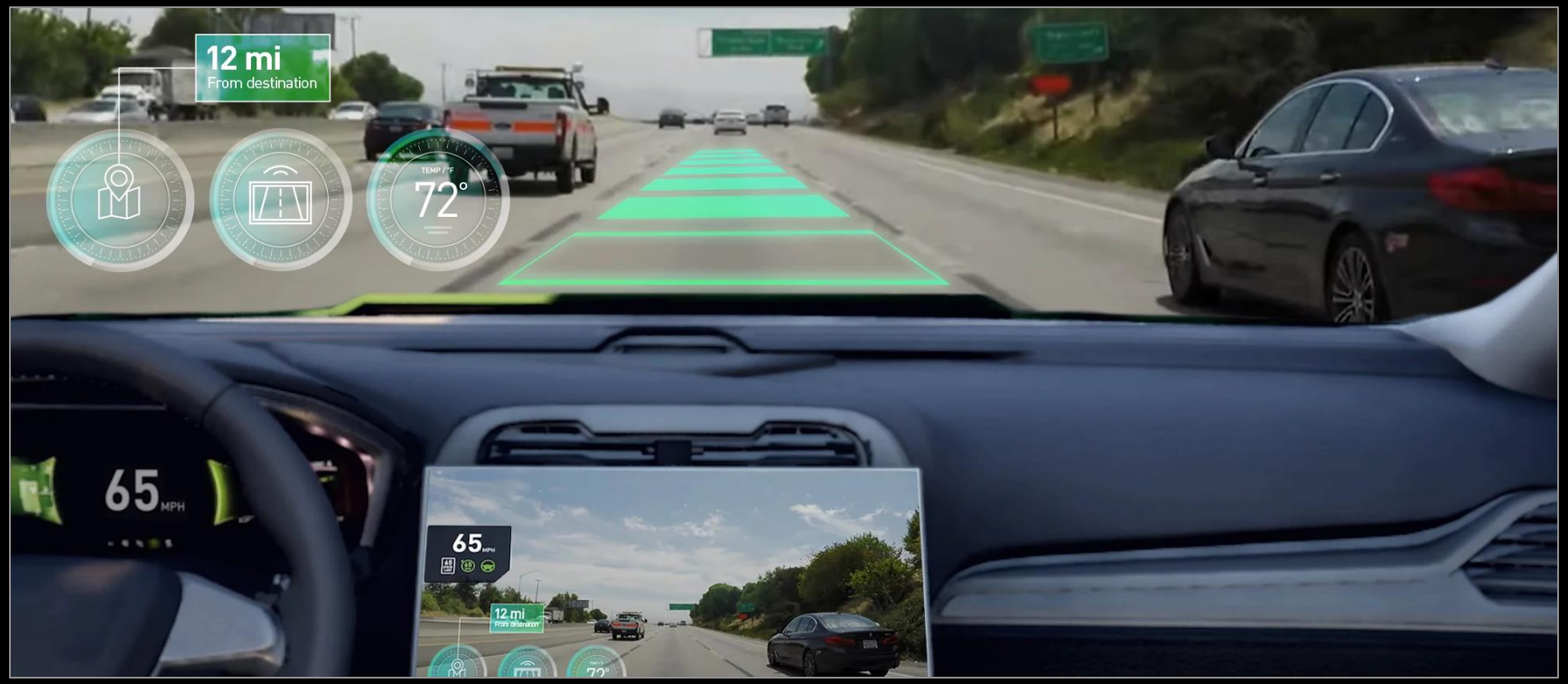
- Graphics is playing a larger role in safety critical industries like Automotive and Avionics.
- OpenGL-SC 2.0 (based on GLES 2.0) was the last safety-critical graphics API. Demands for next generation safety-critical Graphics continues to grow.
- The design of Vulkan SC gives safety-critical application developers detailed control of GPU acceleration in a way that can be rigorously specified and tested to meet safety certification standards such as RTCA DO-178C Level A / EASA ED-12C Level A (avionics); IEC 61508 (industrial), IEC 62304 (Medical), and ISO 26262 ASIL D (automotive).



Khronos Safety-Critical GPU API Evolution

Source: https://www.khronos.org/VulkanSC/

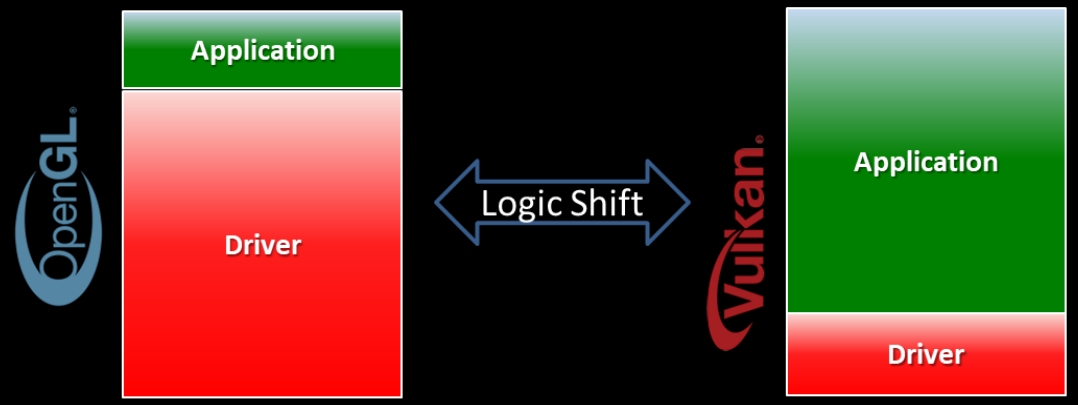
Why Vulkan SC?



Vulkan SC simplifies the Safety certification process enabling manufacturers to deploy advanced graphics capabilities in ADAS systems.

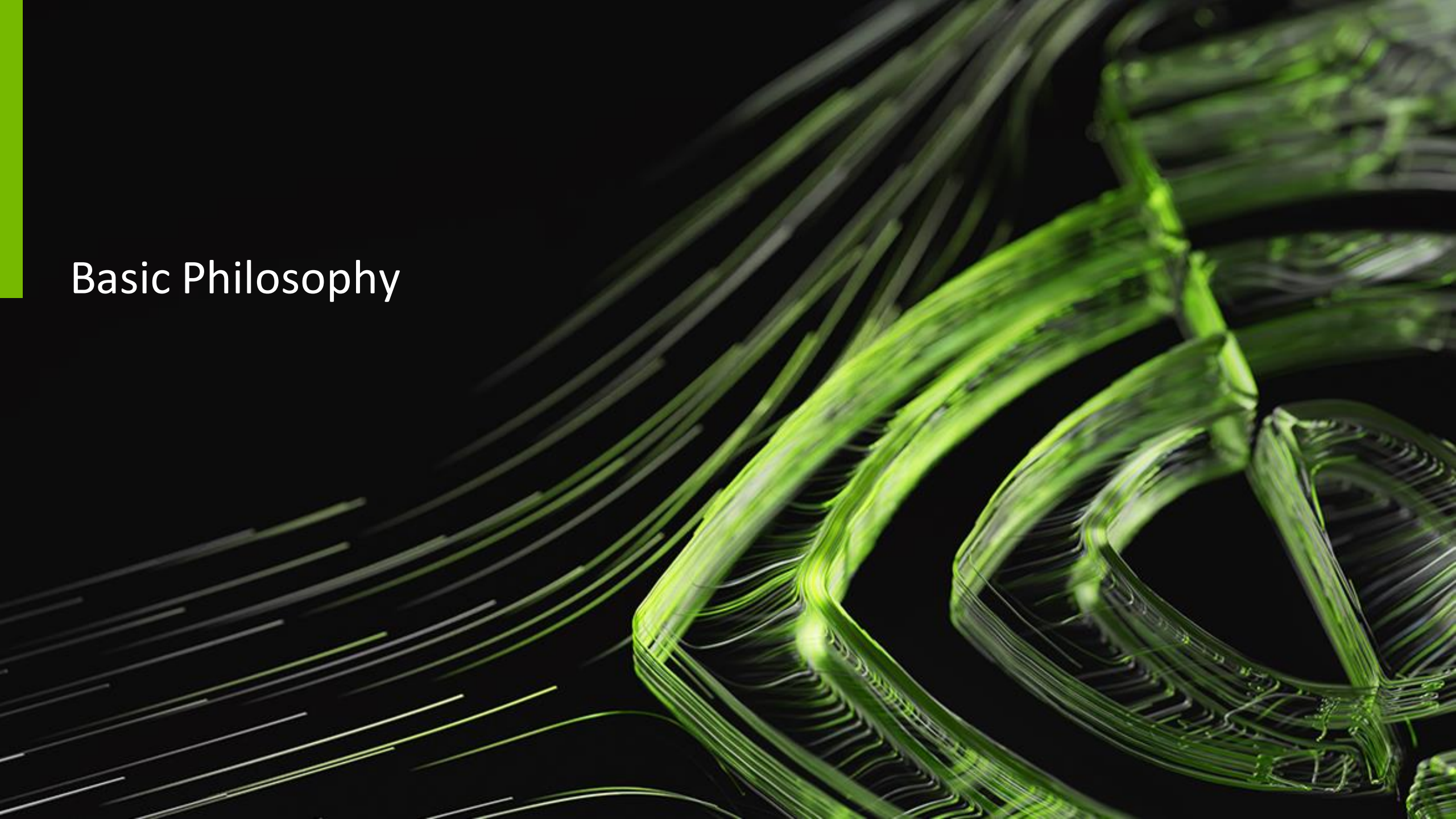
https://developer.nvidia.com/blog/using-vulkan-sc-for-safety-critical-graphics-and-real-time-gpu-processing/

Why Vulkan SC?



- Vulkan 1.2 API provides explicit control to applications over GPU work scheduling, synchronization and resource management.
- Minimal state tracking required in Vulkan implementations compared to GLES
- Using the low level Vulkan API as a starting point for Vulkan SC makes sense from a Safety certification point of view (Less code for IHVs to safety certify).
- Reduced driver complexity.

Basic Philosophy



Philosophy

Overview

Deterministic Execution	• Predictable execution times and results for operations performed through Vulkan SC APIs
Robustness	• Enhanced fault handing, removed ambiguity.
Simplification	• Remove non-essential features to reduce Certification efforts.

Philosophy

Deterministic Execution

Removal of Online compilation

- Online/Runtime compilation is not deterministic. Vulkan SC removes online/runtime compilation.
- Side-effect being the reduction of efforts for IHVs/System Integrators on safety certification

Support for Offline compilation

- In lieu of online compilation, offline compilation support is introduced in Vulkan SC. Pre-compiled binaries are provided as input at runtime by the Vulkan SC application to the implementation. (More on this later).

Philosophy

Deterministic Execution

- Static Memory Allocation
  - Runtime memory allocation results in non-deterministic behavior via potential heap fragmentations, memory allocation errors etc.
  - Application specifies to Vulkan SC implementation/driver the upper bound for number and size of objects of all types that will exist. Implementation allocates memory based on this upper bound during “init” time and Object creation at runtime by application uses the pre-allocated memory.

Philosophy

Robustness

Fault Handling and Reporting

- Application registers functions at device creation which the driver can call if a fault is detected.

Vulkan SC Conformance Test Suite (CTS)

- Leverages existing Vulkan CTS with SC-specific test additions and can be used by IHVs / System integrators to confirm Vulkan SC implementation compatibility.

MISRA C Headers

- Vulkan SC headers are aligned for compliance with MISRA C:2012 (A Software development guideline developed by the MISRA consortium for embedded system code safety, security, portability and reliability and alignment with safety-critical standards).

Philosophy

Simplification

- Depth-stencil resolve made optional
- Shader atomics made optional
- Multiview made optional
- Timeline semaphores made optional
- Removal of sparse memory resources and sparse memory binding support
- Removal of pipeline derivatives
- Removal of host allocation callbacks

Vulkan SC vs Vulkan

Vulkan SC vs Vulkan

- Setup
- Offline Pipelines
- Commands
- Fault Handling
- Object Refresh

Vulkan SC vs Vulkan

- Setup
- Offline Pipelines
- Commands
- Fault Handling
- Object Refresh

Setup

Instance

- Different API version is needed to create a Vulkan SC instance.
- Application info needs to be set differently, as shown below

```
VkApplicationInfo appInfo{};
appInfo.sType              = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pNext              = nullptr;
appInfo.pApplicationName   = "hello_vksc";
appInfo.applicationVersion = 0;
appInfo.pEngineName        = nullptr;
appInfo.engineVersion      = 0;
appInfo.apiVersion         = VK_MAKE_API_VERSION(VKSC_API_VARIANT, 1, 0, 0);
```

Setup

Features & Properties

Vulkan SC supports similar features as Vulkan 1.2.

Exceptions are shader atomic instructions, multiview, depth-stencil resolve, etc. all of which are made optional.

The removals were made in order to reduce driver complexity and verification burden

Vulkan SC-specific device features and properties can be determined via VkPhysicalDeviceVulkan SC10Features and VkPhysicalDeviceVulkan SC10Properties

Vulkan SC adds a new memory heap type - VK_MEMORY_HEAP_SEU_SAFE_BIT

Setup

Device

Applications need to provide information to the driver up-front.

Vulkan SC introduces VkDeviceObjectReservationCreateInfo

Structure is chained to the deviceInfo for device creation.

Setup

Device

- Vulkan SC applications need to provide information to the driver up-front, to do all allocations.
- In order to support this, Vulkan SC introduces `VkDeviceObjectReservationCreateInfo`

```
VkDeviceObjectReservationCreateInfo  devObjectResCreateInfo{};
devObjectResCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_OBJECT_RESERVATION_CREATE_INFO;
devObjectResCreateInfo.pNext = nullptr;
devObjectResCreateInfo.pipelineCacheCreateInfoCount = pipelineCacheCreateInfoCount;
devObjectResCreateInfo.pPipelineCacheCreateInfos    = pPipelineCacheCreateInfos;
devObjectResCreateInfo.pipelinePoolSizeCount = 0;
devObjectResCreateInfo.pPipelinePoolSizes = 0;
devObjectResCreateInfo.layeredImageViewRequestCount = 0;
devObjectResCreateInfo.pipelineCacheRequestCount = 1;
devObjectResCreateInfo.pipelineLayoutRequestCount = 1;
devObjectResCreateInfo.renderPassRequestCount = 1;
devObjectResCreateInfo.graphicsPipelineRequestCount = 1;
devObjectResCreateInfo.computePipelineRequestCount = 0;
devObjectResCreateInfo.descriptorSetLayoutRequestCount = 0;
devObjectResCreateInfo.samplerRequestCount = 0;
// . . .

// Create device
deviceInfo.pNext = &devObjectResCreateInfo;
err = vkCreateDevice(physdevs[0], &deviceInfo, NULL, &dev);
```

Setup

Objects

The VkDeviceObjectReservationCreateInfo structure provides upper bounds on maximum object count.

Objects can be created and destroyed, provided the maximum counts are taken into account.

The Vulkan SC implementation will perform its own host memory allocations. Support for application-provided memory allocation, as supported in base Vulkan, has been removed in Vulkan SC.

Vulkan SC vs Vulkan

- Setup
- **Offline Pipelines**
- Commands
- Fault Handling
- Object Refresh

Setup

Pipeline Compilation

More deterministic compared to online compilation.

Reduced safety certification burden for implementations.

Online shader compilation not supported. Pipeline state needs to be represented offline.

vkCreateShaderModule removed from the API.

JSON as the format for offline pipeline representation. Schema

Offline Pipelines

JSON representation

- JSON as the format for offline pipeline representation. Schema
- Example portion of json shown below.

```
"GraphicsPipelineState" :
{
    "Renderpass" :
    {
        "sType" : "VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO",
        "pNext":"NULL",
        "flags" : "0",
        "attachmentCount" : 2,
        "pAttachments":
        [
        {
            "flags" : "0",
            "format" : "VK_FORMAT_B8G8R8A8_UNORM",
            "samples" : "VK_SAMPLE_COUNT_1_BIT",
            "loadOp" : "VK_ATTACHMENT_LOAD_OP_CLEAR",
            "storeOp" : "VK_ATTACHMENT_STORE_OP_STORE",
            "stencilLoadOp" : "VK_ATTACHMENT_LOAD_OP_DONT_CARE",
            "stencilStoreOp" : "VK_ATTACHMENT_STORE_OP_DONT_CARE",
            "initialLayout" : "VK_IMAGE_LAYOUT_UNDEFINED",
            "finalLayout" : "VK_IMAGE_LAYOUT_PRESENT_SRC_KHR"
        },
        {
            "flags" : "0",
            "format" : "VK_FORMAT_D32_SFLOAT_S8_UINT",
            "samples" : "VK_SAMPLE_COUNT_1_BIT",
            "loadOp" : "VK_ATTACHMENT_LOAD_OP_CLEAR",
            "storeOp" : "VK_ATTACHMENT_STORE_OP_STORE",
            "stencilLoadOp" : "VK_ATTACHMENT_LOAD_OP_CLEAR",
            "stencilStoreOp" : "VK_ATTACHMENT_STORE_OP_DONT_CARE",
            "initialLayout" : "VK_IMAGE_LAYOUT_UNDEFINED",
            "finalLayout" : "VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL"
        }
```

Offline Pipelines

Pipeline Compilation

Pipeline Cache Compiler is used to obtain pipeline cache.

Pipelines are identified at runtime via Pipeline Identifiers specified in VkPipelineOfflineCreateInfo

Pipeline Cache Utility can be used to extract information from cache.

Pipeline derivatives are not supported in Vulkan SC due to the use of read-only offline generated pipeline caches.

Offline Pipelines

PCC

PCC creates offline cache that will be loaded by the runtime application.

PCC takes the SPIR-V shader modules and the JSON state as input and generates pipeline cache.

Necessary to create pipelines offline.

Generates vendor specific offline cache.

Offline Pipelines

Pipeline Pools

Memory for pipelines needs to be specified up-front.

Size provided in VkDeviceObjectReservationCreateInfo struct.

When a pipeline is destroyed, the memory becomes available for re-use if recyclePipelineMemory is supported.

Pipeline Cache Utility can be used to query pipeline size requirements.

Developer needs to obtain the pool size information offline.

Offline Pipelines

Pipeline Creation

In Vulkan SC, pipeline caches are mandated.

The application has to load the cache, specify all required information.

Pipeline state must match the state that was provided to PCC.

VK_PIPELINE_CACHE_CREATE_READ_ONLY_BIT specifies that the cache is read-only.

VK_PIPELINE_CACHE_CREATE_USE_APPLICATION_STORAGE_BIT specifies that the application will maintain the contents of the memory,

```
typedef struct VkPipelinePropertiesIdentifierEXT {
    VkStructureType     sType;
    void*               pNext;
    uint8_t             pipelineIdentifier[VK_UUID_SIZE];
} VkPipelinePropertiesIdentifierEXT;

VkPipelineOfflineCreateInfo pipelineOfflineCreateInfo{};
pipelineOfflineCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_OFFLINE_CREATE_INFO;
memcpy(pipelineOfflineCreateInfo.pipelineIdentifier, identifier, VK_UUID_SIZE);
pipelineOfflineCreateInfo.poolEntrySize = PIPELINE_SIZE;

// Set it to the pipeline's pNext
pipelineCreateInfo.pNext = &pipelineOfflineCreateInfo;
```

Offline Pipelines

Pipeline Creation

- VkPipelineOfflineCreateInfo set as pNext to vkCreate*Pipelines()
- UUID obtained via the JSON.

Vulkan SC vs Vulkan

- Setup
- Offline Pipelines
- Commands
- Fault Handling
- Object Refresh

Commands

Command Pools

- Memory for command pools and buffers need to be allocated by the driver at "init" time.
- VkCommandPoolMemoryReservationCreateInfo needs to be provided by the application.
- commandPoolReservedSize is the number of bytes to be allocated for all command buffer data recorded into this pool.
- commandPoolMaxCommandBuffers is the maximum number of command buffers that can be allocated from this command pool.

```
typedef struct VkCommandPoolMemoryReservationCreateInfo {
    VkStructureType     sType;
    const void*         pNext;
    VkDeviceSize        commandPoolReservedSize;
    uint32_t            commandPoolMaxCommandBuffers;
} VkCommandPoolMemoryReservationCreateInfo;
```

Commands

Command Pools

- Applications are expected to estimate their worst-case command buffer memory usage at development time using vkGetCommandPoolMemoryConsumption and reserve large enough command buffers
- **commandPoolAllocated** is the number of bytes currently allocated from this pool for command buffer data.
- **commandPoolReservedSize** is the total number of bytes available for all command buffer data recorded into this pool.
- **commandBufferAllocated** is the number of bytes currently allocated from this pool for the specified command buffer's data.

```
void vkGetCommandPoolMemoryConsumption(
    VkDevice                                    device,
    VkCommandPool                               commandPool,
    VkCommandBuffer                             commandBuffer,
    VkCommandPoolMemoryConsumption*             pConsumption);


typedef struct VkCommandPoolMemoryConsumption {
    VkStructureType     sType;
    void*               pNext;
    VkDeviceSize        commandPoolAllocated;
    VkDeviceSize        commandPoolReservedSize;
    VkDeviceSize        commandBufferAllocated;
} VkCommandPoolMemoryConsumption;
```

Commands

Command Buffers

Each command recorded into a command buffer has an implementation-dependent size : commandPoolReservedSize.

Command buffers can be allocated and freed in a pool as needed.

Once command buffers are freed, they can once again be allocated from the command pool.

When a command pool is reset, the resources from all command buffers allocated from the command pool are returned back to the command pool.

Vulkan SC vs Vulkan

- Setup
- Offline Pipelines
- Commands
- Fault Handling
- Object Refresh

Fault Handling

Safety Context

- Handling of failures is essential in safety critical contexts.
- In addition to the already existing VkResult enums returned by Vulkan APIs, Vulkan SC adds the following features for enhanced fault event communication between implementations and applications:
  - Optional checks against VUIDs being performed in Vulkan SC implementations.
    - Vulkan SC implementations can return a new VkResult enum "VK_ERROR_VALIDATION_FAILED" to signify validation failures
  - Support for implementations invoking callbacks registered by applications when faults are detected
  - Support for applications querying current faults on demand.
- Performance-critical Vulkan APIs do not usually have error return codes. Vulkan SC implementations can optionally use the Fault Handling features of Vulkan SC to specify runtime faults occurring as part of these APIs.

```
typedef struct VkFaultCallbackInfo {
    VkStructureType             sType;
    void*                       pNext;
    uint32_t                    faultCount;
    VkFaultData*                pFaults;
    PFN_vkFaultCallbackFunction pfnFaultCallback;
} VkFaultCallbackInfo;

typedef void (VKAPI_PTR *PFN_vkFaultCallbackFunction)(
    VkBool32           unrecordedFaults,
    uint32_t           faultCount,
    const VkFaultData* pFaults);


VkFaultCallbackInfo faultCallbackInfo {};
faultCallbackInfo.sType = VK_STRUCTURE_TYPE_FAULT_CALLBACK_INFO;
faultCallbackInfo.pfnFaultCallback = FaultHandlerCallbackFn(); // Callback function

VkDeviceCreateInfo deviceCreateInfo {};
…
…
// Set it to the VkDeviceCreateInfo's pNext
deviceCreateInfo.pNext = &faultCallbackInfo;

// A fault is described as a VkFaultData struct
typedef struct VkFaultData {
    VkStructureType             sType;
    void*                       pNext;
    VkFaultLevel                faultLevel;
    VkFaultType                                     faultType;
} VkFaultData;
```

Fault Handling

Fault handling call-backs

- A new struct VkFaultCallbackInfo specifying a function pointer to be invoked by the implementation when faults occur can be specified during VkDevice creation via extending VkDeviceCreateInfo.

```
VkResult vkGetFaultData(
    VkDevice              device,
    VkFaultQueryBehaviour faultQueryBehaviour,
    VkBool32*             pUnrecordedFaults,
    uint32_t*             pFaultCount,
    VkFaultData*          pFaults
);

// VkPhysicalDeviceVulkan SC10Properties.maxQueryFaultCount specifies
// the maximum number of faults that can be reported by the implementation.
VkFaultData faultData[VkPhysicalDeviceVulkan SC10Properties.maxQueryFaultCount];

// Get the current number of faults available.
VkBool32 unrecordedFaults;
uint32_t faultCount;
VkResult result;
VkFaultQueryBehaviour queryBehaviour = VK_FAULT_QUERY_BEHAVIOUR_GET_AND_CLEAR_ALL_FAULTS;
result = vkGetFaultData(device, queryBehaviour, &unrecordedFaults, &faultCount, nullptr);

// Get the fault data.
result = vkGetFaultData(device, queryBehaviour, &unrecordedFaults, &faultCount, faultData);
```

Fault Handling

Querying Fault Status

- In addition to fault handling call-backs, the number of current faults and fault data can be queried at runtime via the API vkGetFaultData.

Vulkan SC vs Vulkan

- Setup
- Offline Pipelines
- Commands
- Fault Handling
- Object Refresh

Object Refresh

VK_KHR_object_refresh

- Single Event Upset is any inadvertent state change that could cause flipping of bits.
- https://en.wikipedia.org/wiki/Single-event_upset
- Some Safety Critical applications may require this to be supported.
- Host memory typically has error correction enabled for this.
- Device memory may not be SEU safe.
- Explicitly allocated buffer/images can be reloaded by the application.

Object Refresh

VK_KHR_object_refresh

- There could be memory that's allocated by the driver without an explicit allocation by the application.
- This memory is not addressable by the Vulkan SC app.
- VK_KHR_object_refresh extension is for addressing this issue.
- vkGetPhysicalDeviceRefreshableObjectTypesKHR provides objects that have such implicit memory allocations.
- Such memory can be explicitly refreshed via vkCmdRefreshObjectsKHR
- This copies data from an SEU-safe memory.

Vulkan SC Workflow

Workflow

- Several components involved with developing Vulkan SC application.
- Vulkan SC needs a simple way to generate offline JSON pipeline.
- The main components involved in developing the Vulkan SC application are:
  - Vulkan application
  - JSON generation Layer
  - Pipeline Cache Compiler

Workflow

Vulkan application

To develop a Vulkan SC application, start off with a Vulkan application with the same features.

Recommendation is to have both Vulkan and Vulkan SC code co-exist, with the Vulkan SC code compiled out.

This is needed for generating the offline JSON and SPIR-V files.

Workflow

JSON Generation Layer

This is a standard Vulkan layer that can be used to generate the offline pipeline JSON and SPIR-V files.

Set additional environment variables as required by this layer.

Run the Vulkan application in the previous slide with this layer enabled, to obtain the JSON and SPIR-V files.

Obtain the offline JSON and SPIR-V files.

Workflow

PCC

The obtained JSON and SPIR-V files are fed into the PCC to obtain the final pipeline cache.

This pipeline cache is finally used by the Vulkan SC application.

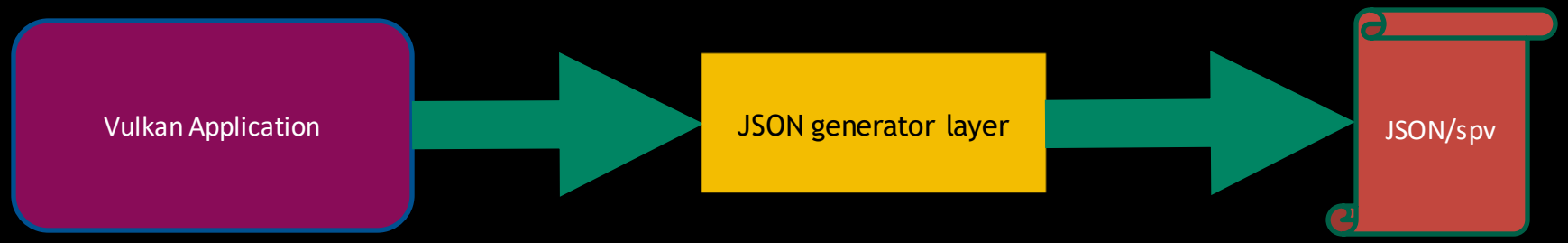
The layer also provides values for VkDeviceObjectReservationCreateInfo that need to be set by the Vulkan SC application.

Workflow

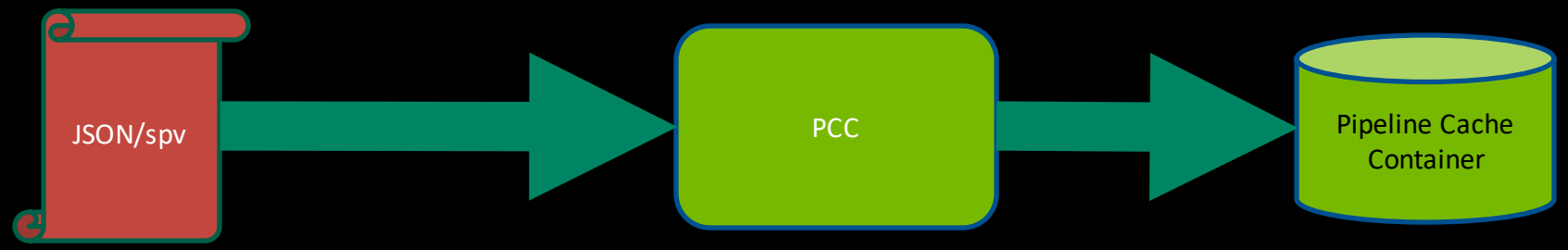
- JSON/SPV needs to be created offline and the cache needs to be obtained.
- Pipeline JSON can be obtained in any way. Recommended way is to use the JSON_generator layer.
- Start with Vulkan app, obtain offline json/spv.
- Compile the cache and feed to VKSC.
- Might need to store different set of caches for different target architectures.

Workflow

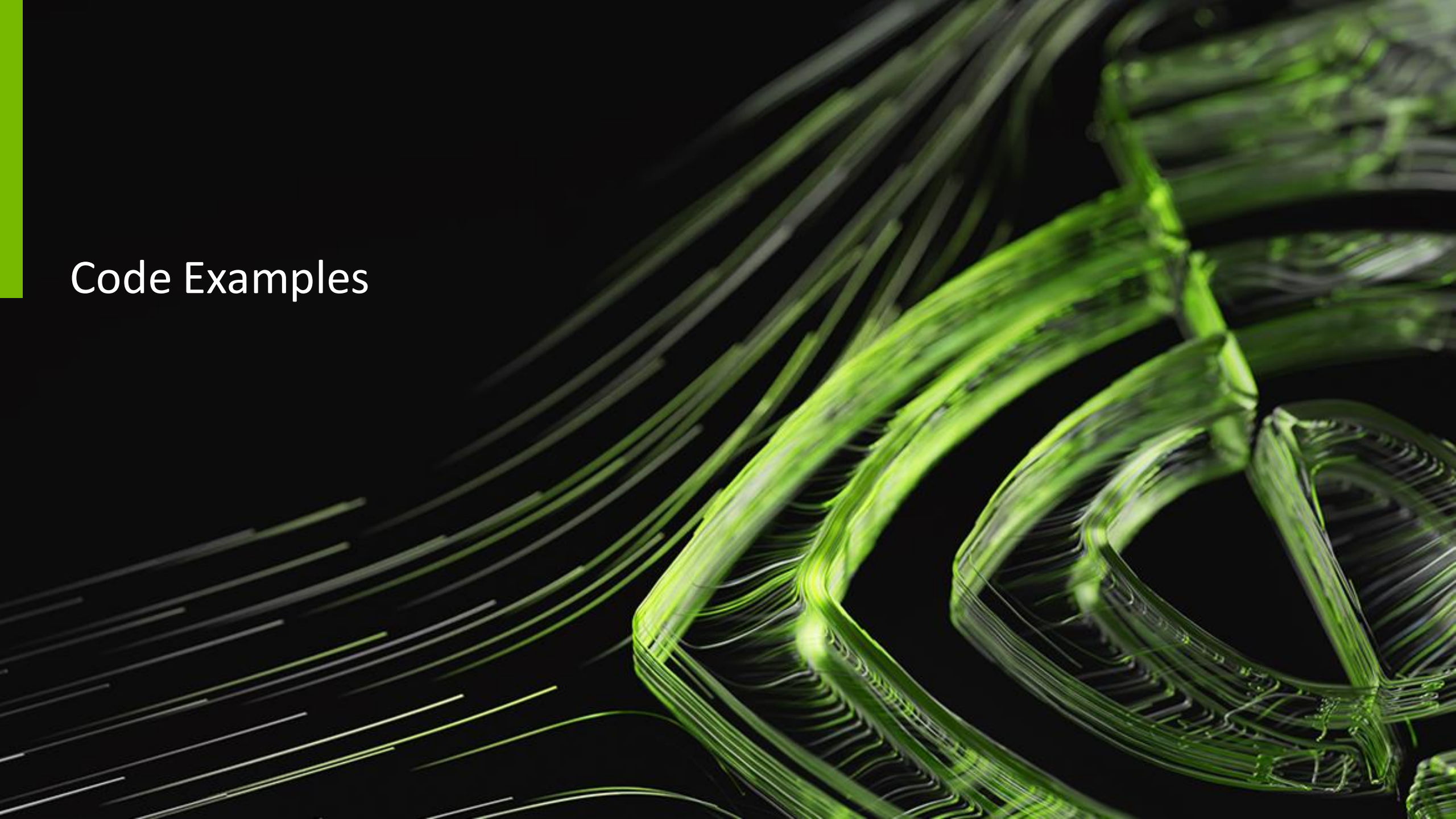
1. Generate JSON file during development



2. Generate pipeline cache container



Code Examples



References

- Vulkan SC home page: https://www.khronos.org/vulkansc/
- Intro blog: https://www.khronos.org/blog/vulkan-sc-overview
- Spec: https://registry.khronos.org/VulkanSC/specs/1.0-extensions/html/vkspec.htm
- Vulkan SC PCUtil: https://github.com/KhronosGroup/VulkanSC-pcutil
- Vulkan SC Loader: https://github.com/KhronosGroup/VulkanSC-Loader
- Vulkan SC Validation layers: https://github.com/KhronosGroup/VulkanSC-ValidationLayers
- Vulkan Samples: https://github.com/SaschaWillems/Vulkan


NVIDIA