1. GPU Validation 2
1.1 Background 5
1.2 Features 6
1.2.1 Analysis 8
1.2.2 Validation 9
1.2.2.1 Resource Address Bounds 10
1.2.2.2 Export Stability 11
1.2.2.3 Descriptor Address Bounds 12
1.2.2.4 Resource Data Race 13
1.2.2.5 Resource Initialization 15
1.3 Implementation 19
1.3.1 GraphicsDevice 20
1.3.2 Diagnostic Allocator 22
1.3.3 JIT-SPIRV 23
1.3.4 Breadcrumbs 24

GPU Validation

This document serves as a how to of GPU validation via the front end / game.

> *The validation of potentially undefined behavior and analysis of performance varying operations on the GPU.*

- Background
  The motivation behind GPU Validation.
- Features
  Validation feature documentation and implementation details.
- *Implementation*
  Implementation notes regarding the GraphicsDevice integration and the GPU Validation layer.

- 1 Starting the game
- 2 Commands
  - 2.1 Begin Report
  - 2.2 End Report
  - 2.3 Export Report
  - 2.4 Flush Report
- 3 Report Types
  - 3.1 Basic
  - 3.2 Concurrency
  - 3.3 Data Residency
- 4 Export Types
  - 4.1 HTML
  - 4.2 CSV
- 5 Basic Usage

1 Starting the game

In order to utilize this feature start the game with the console argument **--graphicsdevice-gpuvalidation**. Note that this alone does not enable validation but enables command hooking for later usage. Note that the first time validating may bloat the startup times due to the shader recompilation, any "new" shaders are automatically cached to the **GPUValidationCache.db** file in the assigned working directory. It is safe to delete this file when this feature is not running.

Counter to the Vulkan validation layers the GPU Validation layers offer interactive frame times while still being able to process upwards of 200-300 million validation messages per second. While this improves the user experience it also allows for more intricate instrumentation such as cross-queue thread safety validation, any further regression in performance would leave such tests void as the test would no longer be an accurate representation of the game.

2 Commands

All commands are available under ***Dev | Graphics | GPU Validation***.

2.1 Begin Report

Begin the validation of GPU operations, there are currently three report types each with their own capabilities, see **Report Types** for more information. Must not already be recording.

Any validation or analysis requires an active "report", the report is the intermediate storage that may later be exported or flushed.

2.2 End Report

End the validation of GPU operations, must be actively recording.

2.3 Export Report

Export a recorded report to file, there are currently two formats available each with their own capabilities, see **Export Types** for more information. Must not be recording.

All report types expose the following validation information:

- Validation message type
  A unique identifier representing the instrumentation / message type
- Shader module
  The parent shader module in which the message was produced
- Source location
  The fine grained [file, {line, column} span] source location of the instrumented instruction
- Source extract
  The respective source extraction of the instrumented instruction

- Function name **[NOT WORKING]**
  The name of the parent function in which the instrumented instruction lies
- Count
  The number of instances in which this unique message was produced

2.4 Flush Report

Dump all validation messages to the default loggers with the appropriate severity. Must not be recording.

3 Report Types

Each report type enables a set of instrumentation modes, see Features for more information.

3.1 Basic

Enables shader instrumentation for Resource Address Bounds, Export Stability and Descriptor Address Bounds.

3.2 Concurrency

Enables shader instrumentation for Resource Data Race.

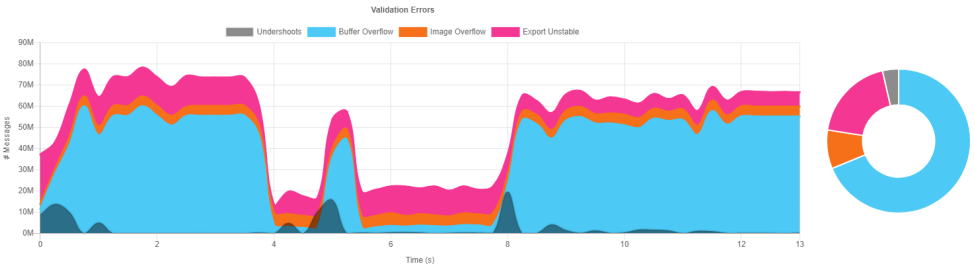
3.3 Data Residency

Enables shader instrumentation for Resource Initialization.

4 Export Types

4.1 HTML

Export to an *inlined* HTML file, requires an active online connection for third party packages. The HTML file also visualizes the number & type of validation messages across time, alongside the latent miss count if enabled.



	Module	Features	Errors (#)	Unique Errors (#)
+	TonemappingHistogramClear.cp	Instrumentation Set Basic	1462	1
+	Deferred_ClusteredLightingIrradiance.fp	Instrumentation Set Basic	3732485	1
+	MotionBlurNeighborMaxFilter.cp	Instrumentation Set Basic	820360	2
+	ASSAO_GenerateAO.fp	Instrumentation Set Basic	8091725	1
+	Deferred_ClusteredLightingLights.fp	Instrumentation Set Basic	15178496	1
+	Deferred_ClusteredLightingLights_FirstPerson.fp	Instrumentation Set Basic	15178496	1
+	VolumetricFogIntegrate.cp	Instrumentation Set Basic	176701440	4

Note from the above that all messages are grouped according to their parent shader module, upon expansion the messages within the module are listed. To get the source extract simply expand once more.

	File	Line	Type	Message	Count
−	TonemappingHistogram.cp	15	Image Overflow	Image address beyond view subresource range	1462

```
HistogramTexture[int2(i, 0)] = 0;
```

+	Deferred_ClusteredLightingIrradiance.fp	Instrumentation Set Basic	3732485	1
+	MotionBlurNeighborMaxFilter.cp	Instrumentation Set Basic	820360	2
−	ASSAO_GenerateAO.fp	Instrumentation Set Basic	8091725	1

	File	Line	Type	Message	Count
−	ASSAO_GenerateAO_ShaderInclude.h	144	Image Overflow	Image address beyond view subresource range	8091725

```
float3 encodedNormal = NormalBuffer.Load( int3( pos, 0 ) ).xyz;
```

4.2 CSV

Exports to a comma separated value sheet, compatible with all sheet editors. Contains mostly the same information as the *inlined* HTML export however is expected to lag behind due to the contraints of the CSV format.

Validation Errors	Count	Type	Message	Module	Source Location	Function Name	Source Extract (Estimation)
	1420	IMAGE_OVERFLOW_AVA	Image address beyond view subresource range	TonemappingHistogramClear.cp	TonemappingHistogram.cp [15:0] - [15:36]	src.main	HistogramTexture[int2(i, 0)] = 0;
	1016010	IMAGE_OVERFLOW_AVA	Image address beyond view subresource range	MotionBlurNeighborMaxFilter.cp	MotionBlurNeighborMaxFilter.cp [19:0] - [19:112]	src.main	float2 velocity = InputTexture[clamp(int2(DTid.xy) int2(u_offset, v_offset), 0, upper_textur
	112890	IMAGE_OVERFLOW_AVA	Image address beyond view subresource range	MotionBlurNeighborMaxFilter.cp	MotionBlurNeighborMaxFilter.cp [82:0] - [82:121]	src.main	OutTextureFloat[DTid.xy] = float4(EncodeVelocityToTexture(NeighborhoodVelocityData.x
	241794	IMAGE_OVERFLOW_AVA	Image address beyond view subresource range	Deferred_ClusteredLightingTunable.fp	LightingShaderIncludes.h [343:0] - [343:62]	main	float4 sh_data = asfloat(SHLookupAddress.Load(load_coords));

5 Basic Usage

In order to begin validating GPU operations a report must be active, to begin a report invoke the following command:

Dev | Graphics | GPU Validation | Begin Report (Basic)

A good indication of an active report is that you may experience an increase of the frame-time alongside some initial stuttering. 🙂

There is no time limit on the report except memory usage, expect a significant increase of both host (CPU) and device (GPU) memory usage on most systems. To end an active report invoke the following command:

Dev | Graphics | GPU Validation | End Report

At this point the report can either be flushed or exported. As mentioned before the flush simply dumps all validation information to the default logger so navigating this information is not a user friendly experience, however, it is useful for determining if there are any errors at all. The exports offer a more user friendly experience and is therefore the recommended option, specifically the HTML export (*requires an active online connection*).

Dev | Graphics | GPU Validation | Export Report (HTML)

Upon completion of the above command the exported HTML file will be automatically opened with the default file handler (commonly Chrome). See the **Export Types** section for more detailed usage.

Background

Throughout the development of a game one often encounters numerous graphical issues, while many are approachable and easily detected there remains two which are commonly known as the bane of existence for graphics programmers.

- Visual Instability
  Visual corruption caused by both defined and potentially undefined operations, such as numeric instability due to insufficient precision.
- Device Instability
  Device (, OS) hangs & crashes caused by potentially undefined operations, such as accessing beyond the buffers subresource byte range. Whille previous APIs attempted the safeguard against such usage newer APIs often do not, particularly Vulkan.

Previously in order to avoid any of the above a strong sense of awareness was required when using any potentially undefined operations GPU side, however that's a ridiculous mental burden that none can satisfy. Validating CPU side issues is less of an issue due to the wide availability of instrumentation tools such as the Intel Parallel Studio (XE), however GPU side instrumentation practically void. While this does directly translate to job security it also means the waste of valuable time tracking down issues with practically no indication of where it originated from.

This leads us to the motivation behind this feature, a layer which attempts to validate potentially undefined behaviour and analysis of performance varying operations on the GPU. While the primary purposes is the validation of such behaviour, it's secondary purpose is the analysis of operations which may result in performance variations of shader invocations. One of such analysis may be the instrumentation of branching in order to produce a global branch coverage report, this is extremely useful for performance analysis on **any** target / API.

This feature is currently exposed as a decoupled Vulkan layer with a lightweight interface on the GraphicsDevice's side. While this does mean that the layer is not available for any other backend (DX12, Orbis, ...) the instrumented operations and analysis are still an accurate representation of other APIs. Maintaining this across all APIs would also not be feasible and would lead to a less capable layer due to time constraints.

Features

This document serves as a basic overview of the validation features available.

Validation

The below features are responsible for the validation of potentially undefined GPU side operations. For a basic description and implementation notes please see the linked individual features.

Resource Address Bounds

`VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_ADDRESS_BOUNDS`

Export Stability

`VK_GPU_VALIDATION_FEATURE_SHADER_EXPORT_STABILITY`

Descriptor Address Bounds

`VK_GPU_VALIDATION_FEATURE_SHADER_DESCRIPTOR_ARRAY_BOUNDS`

Resource Data Race

`VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_DATA_RACE`

Resource Initialization

`VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_INITIALIZATION`

Analysis

The below features are responsible for the analysis of performance varying GPU operations.

(Currently None)

Instrumentation Sets

The features described above are commonly grouped into instrumentation for convenience.

Basic

`VK_GPU_VALIDATION_FEATURE_INSTRUMENTATION_SET_BASIC`

- `VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_ADDRESS_BOUNDS`
- `VK_GPU_VALIDATION_FEATURE_SHADER_EXPORT_STABILITY`
- `VK_GPU_VALIDATION_FEATURE_SHADER_DESCRIPTOR_ARRAY_BOUNDS`

Enables the most common source of undefined behaviour, this has a relatively low performance cost.

Concurrency

`VK_GPU_VALIDATION_FEATURE_INSTRUMENTATION_SET_CONCURRENCY`

- `VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_DATA_RACE`

Enables the detection of invalid threaded behaviour, such as cross-queue resource data races, this has a high performance cost.

Data Residency

`VK_GPU_VALIDATION_FEATURE_INSTRUMENTATION_SET_DATA_RESIDENCY`

- `VK_GPU_VALIDATION_FEATURE_SHADER_RESOURCE_INITIALIZATION`

Enables the detection of (potentially) uninitialized reads, this has a high performance cost.

Wish List

Below is the wish list of instrumentation features yet to be implemented.

- Data Residency
  - Sparse Subresources
- Numeric Stability
  The validation of non-export numeric stability
  - *Zero division*
  - *Underflow & Overflow*
- *Variable Execution*
  The instrumentation of variable execution
  - Indirect Draws
- Coverage
  The analysis of GPU code coverage.
  - Branch Coverage
  - Instruction Coverage
- Data Coherence
  The analysis of sampling / load patterns and their efficiency. Not sure if this is feasible, but it sounds cool 🙂
  - *Texel Coherence*

Analysis

This documents serves as the table-of-contents for all analysis features.

Spoiler: There's nothing!

Validation

This documents serves as the table-of-contents for all validation features.

- Resource Address Bounds
  The instrumentation of raw resource addresses
- Export Stability
  The instrumentation of all export operations
- Descriptor Address Bounds
  The instrumentation of descriptor indexing
- Resource Data Race
  The instrumentation of threaded operations
- Resource Initialization
  The instrumentation of uninitialized resource reads

Resource Address Bounds

The instrumentation of unbounded resource addressing.

Overview

Any operations that allow for unbounded indexing into a GPU side resource are safeguarded and validated against. This applies for:

- **[RW] Buffer**
- **[RW] StructuredBuffer**
- **[RW] Texture#D[Array]**

If the user supplied address is beyond the subresources valid view range value 0 [, 0, ...] is returned instead for read operations. This in turns means that if such an access is responsible for a GPU crash it can be captured.

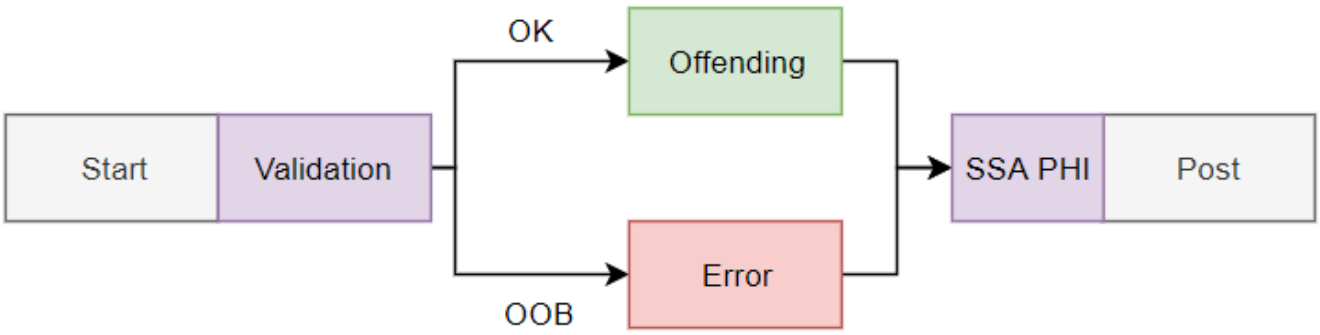
Implementation

Currently the following instruction are instrumented (counter to their name this applies for all of the above resources):

- OpImageWrite
- OpImageFetch
- OpImageRead

The parent instruction block is split into three blocks:

1. Offending Block
   The block in which the original instruction lies
2. Post Block
   The remaining instructions after the original instruction
3. Error Block
   The block producing the validation error



The starting block performs the necessary validation and then conditionally branches to either block, upon read operations the post block then PHI's the final result.

Export Stability

The instrumentation of export operations.

Overview

All export operations that allow for numeric instabilities are validated against. Currently the following export operations are validated:

- **Fragment Export**

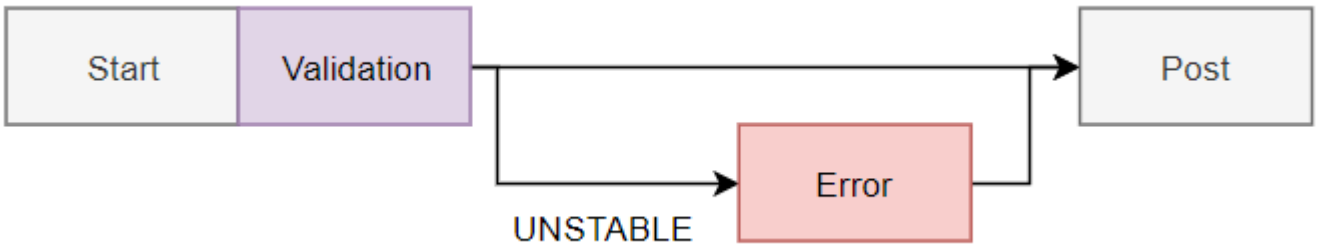
No safeguarding is performed, however would be interesting to avoid cascading issues.

Implementation

As mentioned above currently only fragment exports are validated.

The parent instruction block is split into two blocks:

1. Post Block
   The remaining instructions after the original instruction
2. Error Block
   The block producing the validation error



The starting block performs the necessary validation and then conditionally branches to either block, note that no safeguarding is performed. If the exported value is structured all of it's (recursive) children are traversed. The form of instability is also tracked and reported for all leaf-nodes of the exported value, currently is a (bit) mask of two:

- NaN
  Only applies for FP types
- Infinite
  Only applies for FP types

Note that integral types are not validated as the exported data may be packed to the users specification.

Descriptor Address Bounds

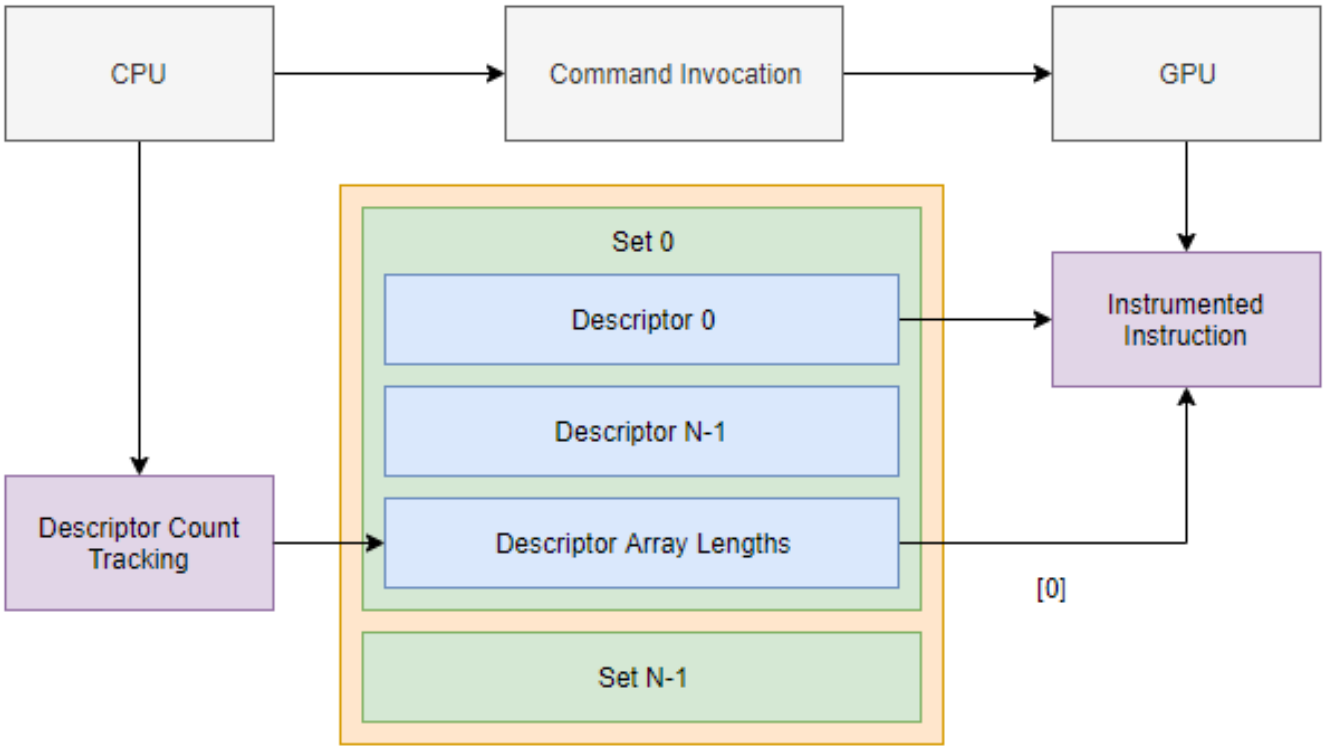
The instrumentation of unbounded descriptor array indexing.

Overview

All descriptor indexing are safeguarded and validated against. This is particularly useful for ray-tracing as it would be the primary source of such usage.

Implementation

Currently the **OpLoad** instruction is instrumented instead of the actual descriptor usages, while this does increase the recompilation time it results in more accurate validation as the descriptor could be manipulated post-load. While SPIRV does contain instructions for determining the length of a runtime array, it only does so for structured runtime arrays which does not apply for descriptors. This in turns means that the HOST has to feed the actual descriptor array lengths to the DEVICE.



The parent instruction block is split into three blocks:

1. Offending Block
   The block in which the original instruction lies
2. Post Block
   The remaining instructions after the original instruction
3. Error Block
   The block producing the validation error

The starting block performs the necessary validation against the HOST fed descriptor array lengths and then conditionally branches to either block, the descriptor index is PHI'd to zero OOB. Note that this PHI has it's own set of issues as the zero'th descriptor may not result in valid usage (raw address OOB's), however is far more stable than an invalid descriptor.

Resource Data Race

The instrumentation of threaded resources across queues.

Overview

Any operations that access resources which may potentially operate across multiple queues are validated against. This applies for:

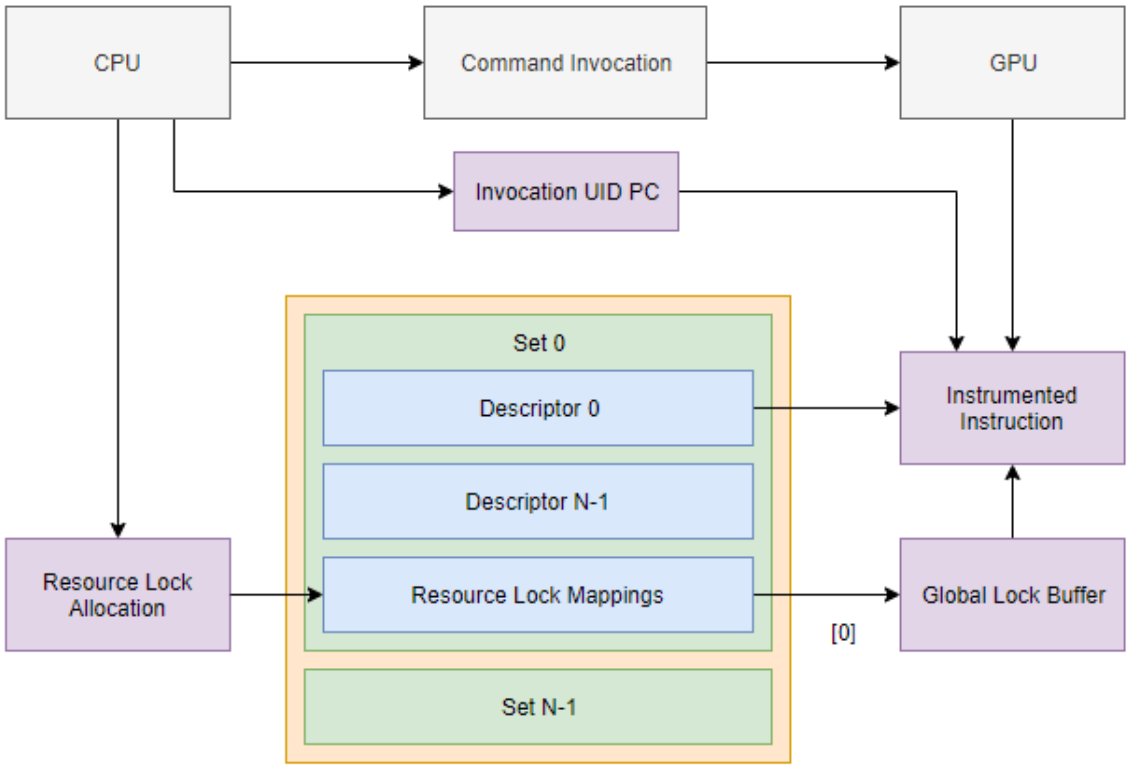
- **[RW] Buffer**
- **[RW] StructuredBuffer**
- **[RW] Texture#D[Array]**

If a race condition is detected no safeguarding is performed (subject to debate).

Implementation

In order to accurately track the resources across queues each resource requires a global unique "lock" state, this in turns leaves us two issues:

1. Descriptor to lock UID mapping
   How is a descriptor index mapped to a global lock UID
2. Lock representation
   How is such a lock represented? What about resources within the same shader?



The descriptor lock UID mapping is performed via descriptor set injection, a new uniform buffer is injected which performs descriptor binding (within the set) to global lock UID mapping. The lock UID allocation is performed CPU side where the appropriate subresource is taken into account. The lock value is fed via push-constants and is a unique value representing the current command invocation.

Note that RenderPasses are also considered, the lock values are (forcibly) written before vkCmdBeginRenderPass and after vkCmdEndRenderPass.

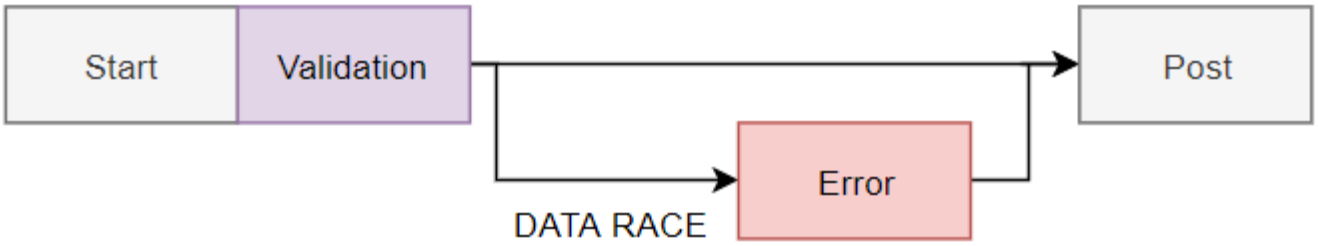
Currently the following instruction are instrumented (counter to their name this applies for all of the above resources):

- OpImageSampleImplicitLod
- OpImageSampleExplicitLod
- OpImageSampleDrefImplicitLod
- OpImageSampleDrefExplicitLod
- OpImageSampleProjImplicitLod
- OpImageSampleProjExplicitLod
- OpImageSampleProjDrefImplicitLod
- OpImageSampleProjDrefExplicitLod
- OpImageFetch

- OpImageGather
- OpImageDrefGather
- OpImageRead
- OpImageWrite

The parent instruction block is split into two blocks:

1. Post Block
   The remaining instructions after the original instruction
2. Error Block
   The block producing the validation error



Read Operations

Upon any immutable operation the global lock value is fetched for the particular resource and compared against zero (not locked), then conditionally branches to error if not zero.

Write Operations

Operations that modify the resource perform a strong atomic compare exchange against the global lock address with the comparator being zero and the value being the invocation UID value. It is deemed thread safe if the previous lock value (return value) is either zero or the current invocation UID, it is conditionally branched to error if neither apply.

Notice

Note that if the shader has an address pattern which does not result in collisions across multiple concurrent invocations this will produce false positives.

Resource Initialization

The instrumentation of uninitialized sub-resource reads.

Overview

Any operations that can access potentially uninitialized sub-resources are validated against. This applies for:

- **[RW] Buffer**
- **[RW] StructuredBuffer**
- **[RW] Texture#D[Array]**

If an uninitialized read is detected no safeguarding is performed (subject to debate).

Upon any write operation (shader, command queue, system write) the respective sub-resource range is marked as initialized.

Implementation

The goal is to track the initialization states (initialized or uninitialized) per sub-resource of all resources in a performant and conservative manner, this in turn leaves us with the following issues:

1. Sub-resource initialization mask representation
   How is this mask represented? What are its constraints?
2. Descriptor to sub-resource initialization mask UID mapping
   How do we perform the mapping from a descriptor binding to an external mask storage?
3. Non 4-byte aligned in-command-buffer mask writes
   For in command buffer initialization (such as vkCmdCopyImage) how do we write (bitwise or) masks that are not 4 byte aligned?
4. Host resource initialization
   System residing resources may be initialized outside a command buffer, how do we write the DEVICE side mask and have it be visible on all queues safely?
5. Block segmentation strategy

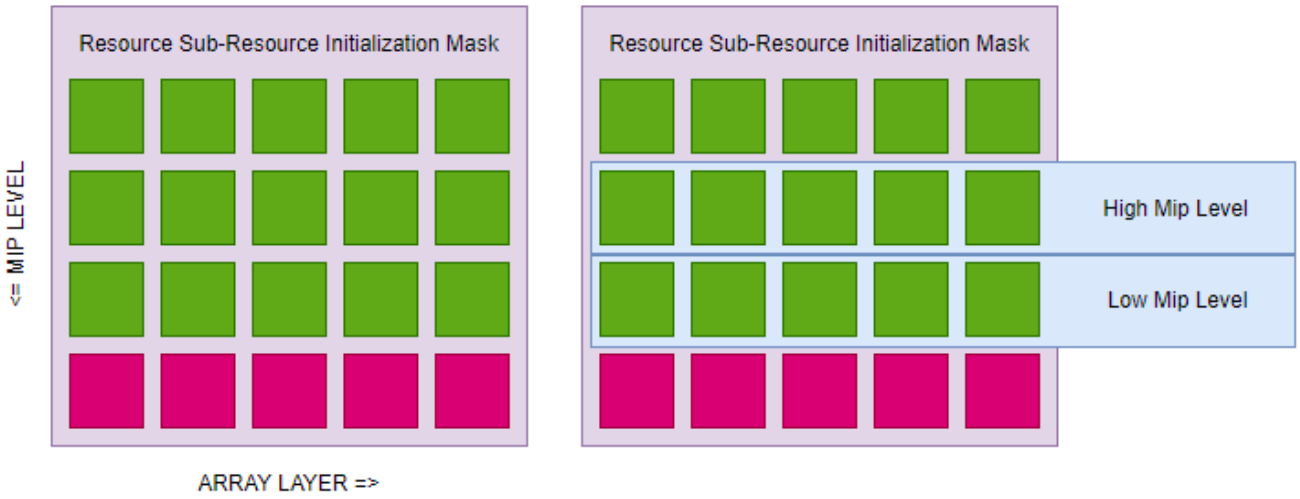
Sub-Resource Mask Representation

Due to the performance constraints it was chosen to represent the sub-resource states of a resource as a single 32b bit mask, this ensures that only a single load is required during sub-resource read validation.

However, while most sub-resource read ranges are uniform per descriptor some instructions do not follow this pattern, such as:

- ImageSampleExplicitLod
- OpImageSampleDrefExplicitLod
- OpImageSampleProjExplicitLod
- OpImageSampleProjDrefExplicitLod
- OpImageFetch

Notably the above instructions may alter the source / destination mip, also known as lod, level. Because of this the sub-resource mask (the mask of bits we expect to be set / "initialized") must account for this. It is also important to note that the mip level is a floating point (with fetch being an integer), this means that two sub-resource ranges may be conditionally accessed. In order to mask out the parent sub-resource mask without relying modifying the control flow the following bit layout was chosen:



Because the mip level is represented as the "Y axis" it simplifies masking out individual mip levels as the required mask bits are completely sequential. However, if the array layer was instead represented as the y axis the bit pattern of the mip mask would be sparse and difficult to setup in a performant manner. The sub-resource bit mask of a single mip level is constructed as following (ArrayLayerCount is represented as the fixed "X axis" width):

```
MipSRMask(MipLevel) = (~0U >> (32 - ArrayLayerCount)) << MipLevel * ArrayLayerCount
```

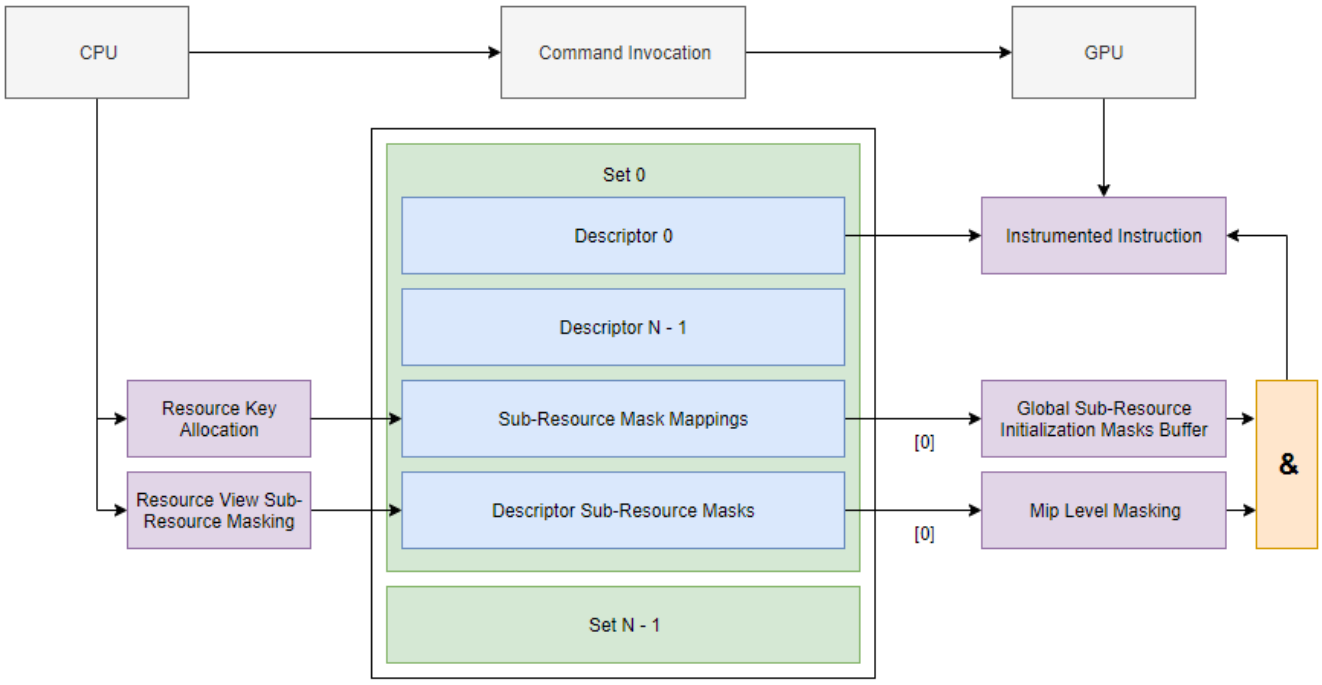
With the above being a single mip level, the parent mask is modified as such:

```
SRMask &= MipSRMask(Floor(MipLevel)) | MipSRMask(Ceil(MipLevel))
```

While this design is performant is has the hard constraint that the total sub-resource count (**ArrayCount * Mip Count**) must not exceed 32, this is seen as acceptable.

Descriptor Mapping

The descriptor sub-resource mask UID mapping is performed via descriptor set injection, a new uniform buffer is injected which performs descriptor binding (within the set) to global sub-resource mask UID mapping. The sub-resource mask UID allocation is performed CPU side and only takes the decayed resource into account (VkImage, VkBuffer), the sub-resources are accounted for in a secondary uniform buffer which holds the sub-resource masks of all descriptors bound within the same set. Upon instruction instrumentation both the descriptor key mapping and sub-resource mask are loaded.



Note that the host fed resource view sub-resource masking can account for sparse bit patterns whereas the GPU side on-the-fly mip level masking cannot, hence why the layout of the sub-resource mask is so important. The higher 6 bits of the sub-resource mask mappings contain the ArrayLayerCount for the particular descriptor.

Command Buffer Mask Writes

Sub-resource (assumed) initialization may occur outside of a shader during any of the following commands:

- vkCmdBeginRenderPass
- vkCmdCopyBuffer
- vkCmdCopyImage
- vkCmdBlitImage
- vkCmdCopyBufferToImage
- vkCmdCopyImageToBuffer
- vkCmdUpdateBuffer
- vkCmdFillBuffer
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdClearAttachments
- vkCmdResolveImage

This in turns means that the resource in question needs to be initialized during one of the above commands in-command-list.

Buffers

The sub-resource masks for all buffers assume a single subresource, this means that any buffer "initialization" can be a simple fill operation with the appropriate offset into the global sub-resource initialization mask buffer.

Images /Views

As descriptor image and image views may have both sparse bit patterns and non 4-byte aligned bit patterns (command buffer side writes require 4-byte alignment) therefore an immediate compute kernel is required for such writes. The kernel performs the following operation:

```
InterlockedOr(GlobalSRMaskBuffer[ResourceKeyUID], SRMask);
```

After the kernel has finished executing it must restore the previously bound descriptor sets. Simply tracking the bound sets and re-binding them did not prove sufficient as they may have decayed (automatic de-binding due to incompatibility with the current pipeline layout), therefore a manual descriptor set layout hash was computed during creation and compared against to avoid this issue.

However, another issue is that the resource may have been "initialized" on a dedicated transfer queue with no compute capabilities. One potential implementation would be to cheat and mark the entire sub-resource mask as initialized but this will prevent accurate diagnostic. Therefore when GPU validation is enabled all dedicated transfer queue families are emulated and proxied to a dedicated, to avoid scheduling contention issues, compute queue. This means that intricate sub-resource mask manipulation is now possible for all GPU operations.

Host Resource Initialization

Some commands may initialize (possibly indirectly) a resource outside of a command list for system resident resources, this may occur during any of the following commands:

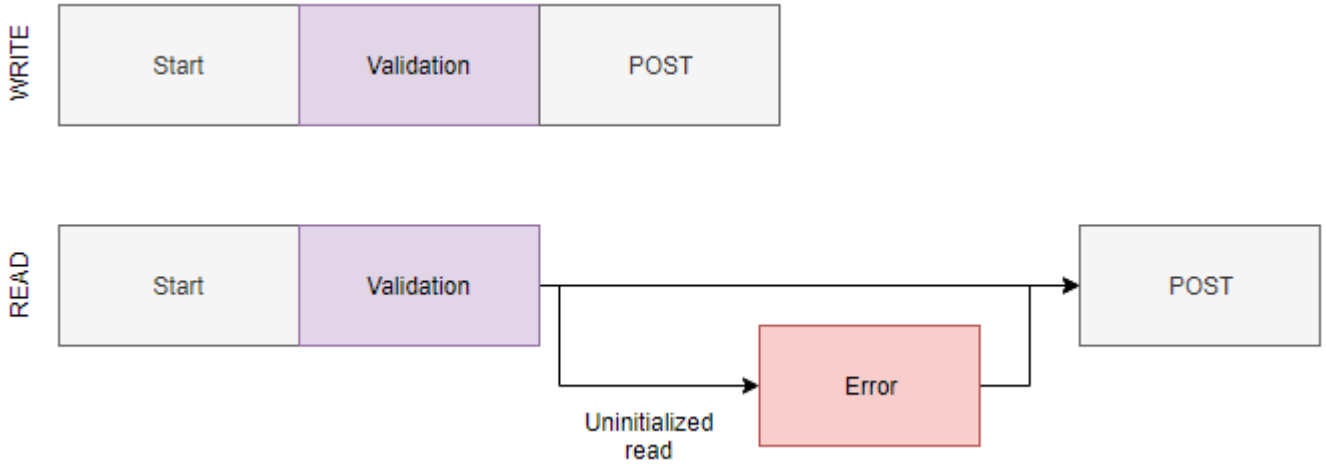
- vkMapMemory
- vkBindBufferMemory
- vkBindImageMemory

One potential solution would be to submit the sub-resource mask initialization commands on the non-waiting queue, but this could potentially result in false positives if a neighbouring queue had no scheduling constraints. In order to avoid this the initialization commands are repeated per available queue and submitted alongside standard submissions during vkQueueSubmit. Repetition of initialization commands are thread safe so this is not an issue.

Block Segmentation Strategy

The parent instruction block is split into two blocks for read operations, however, for write operations no segmentation is required.

1. Post Block
   The remaining instructions after the original instruction
2. Error Block
   The block producing the validation error



Write Operations

Operations that modify the resource perform an atomic bitwise or on the global sub-resource mask with the local sub-resource mask.

Read Operations

Upon an operation which reads potentially uninitialized data from a sub-resource the global sub-resource mask is loaded and compared against the local sub-resource mask as following:

```
(GlobalSRMaskBuffer[ResourceKeyUID] & LocalSRMask) == LocalSRMask
```

The validation block is conditionally branches to the error block if any of the bits deviate (i.e. the resulting value is false), from which a validation error is exported.

Implementation

This documents serves as the table-of-contents for all implementation specific details.

- GraphicsDevice
  The ctg-GraphicsDevice side integration of the GPU Validation layer.
- *Diagnostic Allocator*
  The design of the diagnostics allocator.
- *JIT-SPIRV*
  The design of the just-in-time recompilation of SPIRV modules
- *Breadcrumbs*
  The process at which command buffer states are tracked and reconstructed during message filtering

GraphicsDevice

This document serves as the documentation for the GraphicsDevice side integration of the GPU Validation layer.

Initialization

The GPU Validation layer is exposed as a instance layer and device extension, registration of both is required.

Instance Registration

The instance layer name is exposed as **VK_LAYER_AVA_gpu_validation_NAME**, as with any other layer this is passed in during instance creation. Note that as with the shader reloading layer external debuggers, such as RenderDoc, may not play nice with this enabled despite the specification conformance. However, this can be safely enabled alongside the standard validation layers.

Device Registration

The device extension is exposed as **VK_AVA_GPU_VALIDATION_EXTENSION_NAME**, query for support vkEnumerateDeviceExtensionProperties first to ensure support. The device extension requires an extensive creation structure describing the limits and usage of the GPU Validation layer, this is exposed via **VkGPUValidationCreateInfoAVA**, as with any other extension structure the appropriate, **VK_STRUCTURE_TYPE_GPU_VALIDATION_CREATE_INFO_AVA**, structure type is required. This structure is chained to the device creation info and contains the following set of configurables:

- Features
  The globally enables feature (bit) mask, each validation message type is covered under a single feature.
- UserData
  The user defined data passed alongside all callbacks.
- MessageCallback
  The callback for flushed messages
- LogCallback
  The callback for any generic logging, severity alongside source level granularity is provided.
- LogSeverityMask
  The mask of messages with given severity to log.
- AsyncTransfer
  Enables the use of a dedicated asynchronous transfer queue for the DEVICE HOST transfer of validation messages.
- LatentTransfer
  Allows the async transfer to estimate the number of validation messages to transfer, significantly reduces the bus load however may result in missed messages.
- CommandBufferMessageCountDefault
  The initial validation message capacity of a command list, grows to <CommandBufferMessageCountLimit>
- CommandBufferMessageCountLimit
  The maximum amount of validation messages a single command list may produce.
- ChunkedWorkingSetByteSize
  The internal allocation size for general purpose usage.
- ThrottleThresholdDefault
  If the GPU is emitting validation messages faster than the CPU can process the frame may need to be throttled in order to let the CPU catch up, this initial threshold denotes the number of frames the GPU can initially run ahead of the CPU, grows to <ThrottleThresholdLimit>.
- ThrottleThresholdLimit
  This threshold denotes the maximum amount of frames the GPU can run ahead of the CPU.
- ShaderCompilerWorkerCount
  The number of workers to spawn for shader compilation
- PipelineCompilerWorkerCount
  The number of workers to spawn for pipeline compilation
- CacheFilePath
  The intermediate cache file path, extension does not matter.
- StripFolders
  Should the folders be stripped of all source level paths? This is recommended.

Function Querying

The following functions may be queried via the device:

- "**vkGPUValidationCreateReportAVA**"
  Creates a new report instance
- "**vkGPUValidationDestroyReportAVA**"
  Destroys a report instance, must not be recording.
- "**vkGPUValidationBeginReportAVA**"
  Begin recording to a report, must not be recording.
  *Does not clear the report, see <"**vkGPUValidationFlushReportAVA**">*
- "**vkGPUValidationGetReportStatusAVA**"
  Returns the current state of a report
- "**vkGPUValidationDrawDebugAVA**"
  Utilizes the 'IDebugRenderer' to print basic information about the active report and its contents
- "**vkGPUValidationEndReportAVA**"
  End the recording to a report, must have been previously recording.

- "**vkGPUValidationPrintReportSummaryAVA**"
  Print the summary of a report, this does not include the individual validation messages.
- "**vkGPUValidationPrintReportAVA**"
  Print the entire report to the assigned LogCallback and MessageCallback.
- "**vkGPUValidationExportReportAVA**"
  Export the report to a specified format.
- "**vkGetReportInfoAVA**"
  Gets the report info including its exported messages
- "**vkGPUValidationFlushReportAVA**"
  Clear a report, must not be recording.

Shader Module Creation

During the creation of shader modules an optional **VkGPUValidationShaderCreateInfoAVA** extension structure can be passed in which defines the following:

- Name
  The name of the module to be used for all validation messages

Pipeline Creation

During the creation of pipelines an optional **VkGPUValidationPipelineCreateInfoAVA** extension structure can be passed in which defines the following:

- Name
  The name of the pipeline to be used for all validation messages
- FeatureMask **[Not Implemented]**
  Mask of features to enable on this pipeline

Diagnostic Allocator

<TODO>

JIT-SPIRV

<TODO>

Breadcrumbs

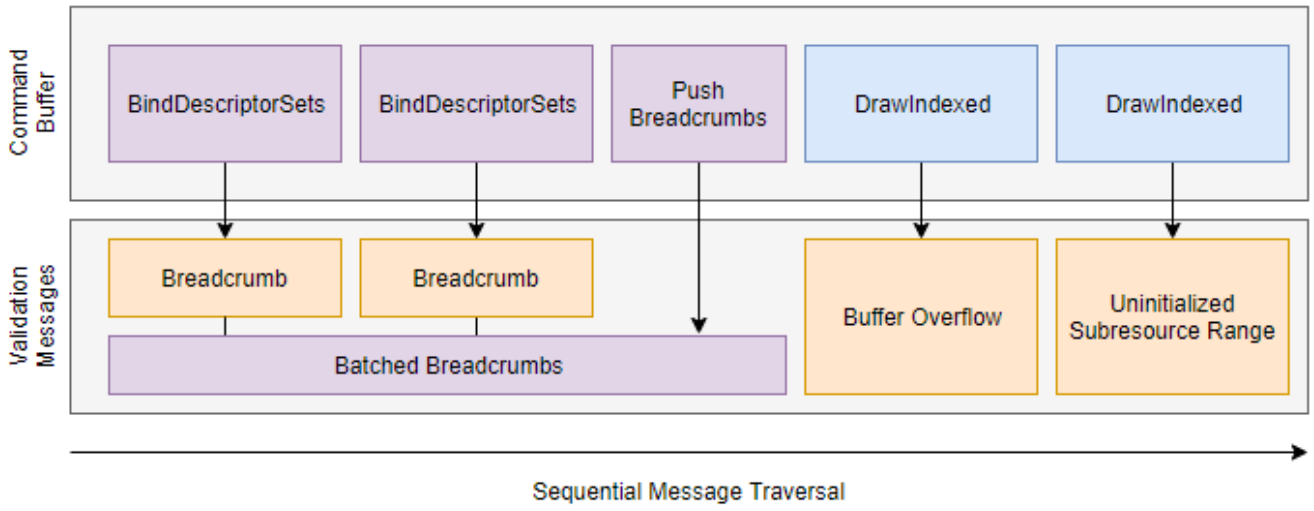
Breadcrumbs are the process at which command buffer information is reconstructed serially during message filtering, almost any information can be reconstructed with appropriate tracking.

Without the use of breadcrumbs most validation errors are only able to reconstruct a sparse set of data from which the end user can try to backtrace the program. While this may be sufficient for decoupled and low frequency GPU tasks it gets increasingly complicated to backtrace errors found within common functionality.

With the features introduced by this document each validation error may report exactly which descriptor specifically during the recording of the error is of interest. This is all done without modifying any of the exported pass specific message data as only few bits remain which would prove insufficient.

Overview

In order to record and apply the appropriate state reconstruction during message filtering the breadcrumbs, which represent a state change, are inserted alongside standard validation messages. Since the GPU side message queue is already serial and sequential this ensures that a specific message can query any tracked state without error.



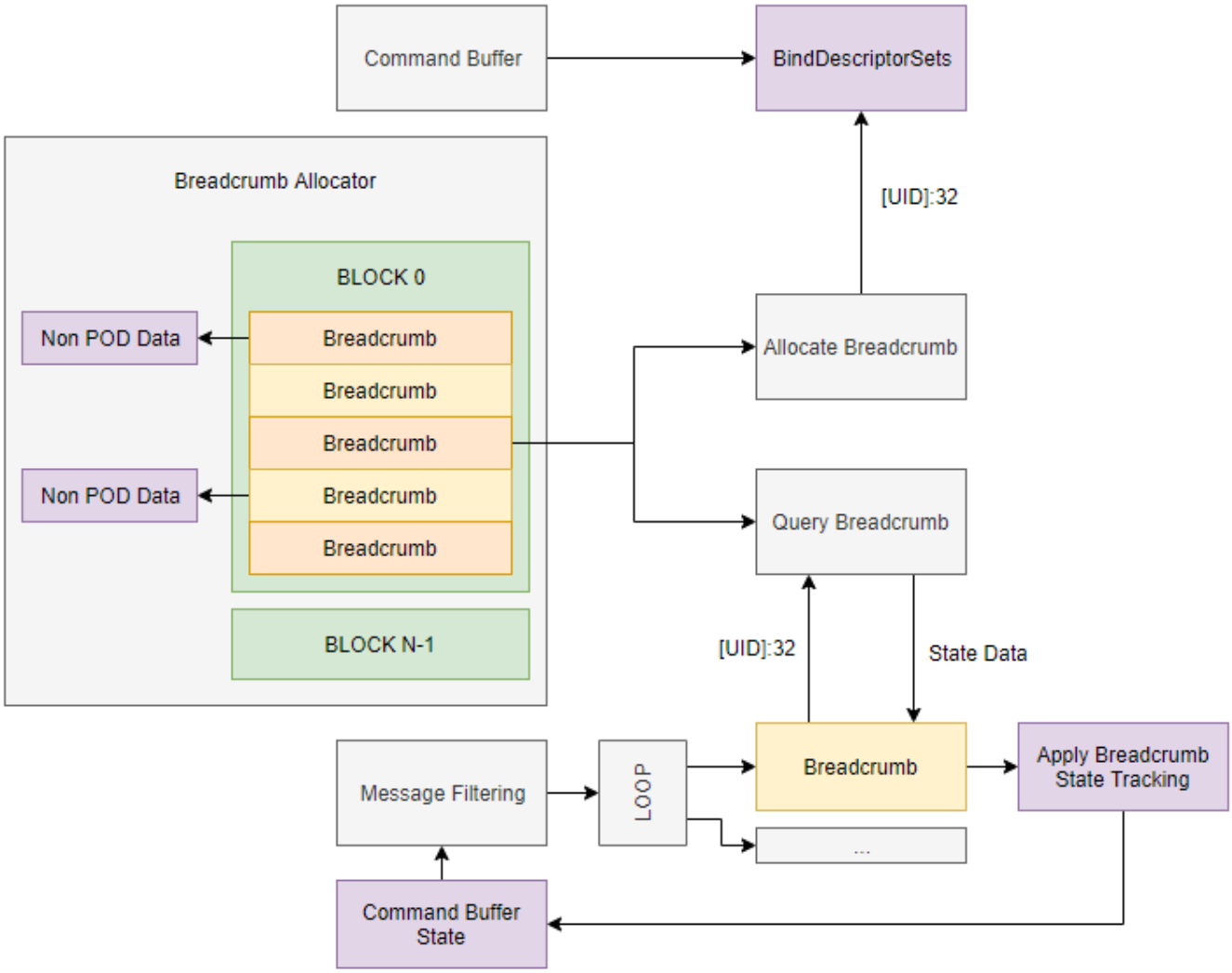
During message filtering iteration if such a breadcrumb is encountered the state change is applied to a temporary command buffer state, any diagnostics pass can query from this state.

However, most validation messages are already short of space (with most only having a few bits to spare) so adding additional information here is infeasible. The payload of each message could be increased but since we are already over-taxing the PICE bus this too is not a good choice. Instead the shader location extracts are optionally associated with a specific (set, binding) descriptor binding which can be queried later during message filtering. Once the descriptor binding has been queried the "currently" (at the point of the exported message) bound descriptor is extracted from the command buffer state.

This can also be expanded to any other command buffer state that requires tracking, however the descriptor data was found sufficient for now.

Implementation

As each message payload is limited to 4 bytes the exported data is a single index from which the host local breadcrumb data can be found. The hosted breadcrumb data can represent any command buffer state change such as descriptor sets (currently the one change recorded). Since each breadcrumb may host non-pod data which may be expensive to copy the breadcrumbs follow a blocked allocation strategy that avoids expensive reallocations.



As descriptor sets can be updated post-usage and message filtering is complete asynchronous the descriptors themselves need to be copied and tracked manually. Incorporating this into the host breadcrumb data itself would be inefficient as it would either require an allocation upon each breadcrumb allocation, or a worst case scenario of all breadcrumbs hosting the worst-case possible descriptor storage space. Instead, the actual descriptor data is indirect and hosted in a decoupled cache which is reused when appropriate.

Injection Kernel

The DirectXShaderCompiler was insufficient as it cannot represent runtime arrrays in accordance with our SPIRV requirements. Instead the disassembled SPIRV was hand written and then both assembled using `spirv-as` and then validated using `spirv-val`.

For record the kernel for writing (batched) breadcrumb data can be found here:

http://opengrok/xref/ctg-projects/vulkan_layers/shaders/gpu_validation/Passes/BreadcrumbWrite.spirvdis