Vulkan GPU-Assisted Validation

Karl Schultz, LunarG
Tony Barbour, LunarG
February 2021 - Revision 5

Table of Contents

Introduction

The Vulkan validation layers perform a variety of API usage checks during application execution. These checks verify that the application is using the API correctly by performing stateless parameter checking, object lifetime tracking, object state validation, and various other checks. The layers perform these checks on the CPU as the Vulkan application executes and provide valuable information about Vulkan API usage to the developer.

However, much of an application's activity is on the GPU, where the CPU-based validation layers have little visibility. This paper explains the GPU-assisted validation implementation and how a developer can use it. This feature's design is intended to allow the developer to use nearly the same validation layer workflow as before.

What is GPU-Assisted Validation?

In general, GPU-assisted validation involves using the GPU to check for API usage errors at shader execution time. These on-GPU checks can't always find general shader program logic errors but can detect certain run-time problems like out-of-bounds (OOB) indexing into descriptor arrays and accessing invalid descriptors.

The entire GPU-assisted validation process also consists of communicating the results of the GPU's checks back to the CPU so that the layer can report any violations to the user.

This paper discusses several types of GPU-assisted validation that have been implemented in the validation layer.

Bindless Descriptor Access

This type involves checking the indexing into an array of descriptors, otherwise known as "bindless" descriptor access.

The following example shows what is meant by bindless descriptor access.

```
layout (set = 0, binding = 0) uniform sampler2D normal_tex;
layout (set = 0, binding = 1) uniform sampler2D bindless_tex[6];
layout (set = 1, binding = 0) buffer uniformBuffer_t
{
   uint tex_index;
} uniformBuffer;
```

Descriptor set 0 contains two bindings. The first binding has a single descriptor and can be accessed without ambiguity. The second binding is an array of like descriptors and simply

referencing it with `bindless_tex` is ambiguous and bindless because the shader compiler does not know which of the application's resources to attach to this bind point. A variable like `tex_index` is used to resolve this ambiguity at run-time by indexing into the array.

The index used by the shader can be a variable whose value is not known until the shader executes. Therefore, the index into the descriptor array cannot be checked to see if it is in bounds until shader execution time.

The implementation checks only descriptors for images and texel types as it is fairly common for applications to present an array of these types of resources to a shader.

Descriptor Indexing

In April 2019, GPU-assisted validation code was updated to include validation for scenarios that arise when the `VK_EXT_descriptor_indexing` extension is enabled.

The `VK_EXT_descriptor_indexing` extension allows applications increased flexibility when initializing or updating descriptors. Without this extension, applications need to have all descriptors updated before binding a descriptor set to a pipeline. But when applications do use this extension, validating the descriptors becomes more difficult on the CPU side.

The descriptor indexing extension allows for the following scenarios:

- **runtimeDescriptorArray** - The sizes of descriptor arrays can be determined at runtime rather than at shader compile time

- **descriptorBindingVariableDescriptorCount -** An array at the last (highest) binding point can have a variable descriptor count from set to set

- **descriptorBindingPartiallyBound** - A descriptor can be partially bound and only those elements accessed by the shader need to have been written

- **descriptorBindingSampledImageUpdateAfterBind -** Descriptors can be written after the descriptor set has been bound, but before the command buffer is submitted to a queue

The new validation code detects out of bounds indexing of descriptor arrays and use of unwritten descriptor elements when the above features are being used. It does this by adding another debug buffer, an input buffer, from the perspective of the shader instrumentation code. This input buffer tells the validation code in the shader the size of all bound descriptor arrays as well as the write state of all descriptor elements within those arrays. The instrumented shader code then checks all descriptor array indexing against the supplied sizes, and all element

accesses against the element write states to flag out of bounds indices and use of unwritten elements. The results are then processed and reported by the validation layer.

Note that descriptor indexing validation uses more device memory in supplying array sizes and element write states to the validation code in the shader via the input buffer. In particular, the size of the input buffer is determined by the number of bound sets, the sum of the largest binding number in each set, and the sum of the number of elements in all of the sets. Using sparse binding numbers that result in a set with a large maximum binding number will cause the input buffer to be larger than it would with more densely packed binding numbers. So using densely packed binding numbers is encouraged as a best practice when using GPU-assisted validation with descriptor indexing enabled.

Buffer Device Address Validation

In September 2019, GPU-assisted validation code was updated to include validation for buffer accesses with the `VK_EXT_buffer_device_address` extension enabled.

The `VK_EXT_buffer_device_address` extension allows an application to retrieve a device address from the driver for any given buffer they have created. It can then use that address in a shader to access that buffer directly, without the need for binding descriptors.

The new validation code detects out of bounds accesses using addresses that were obtained from calls to vkGetBufferDeviceAddressEXT. It does this by adding an input buffer (separate from the input buffer used for descriptor indexing validation) that lists all addresses obtained from vkGetBufferDeviceAddressEXT along with the stated size of the associated buffer. It instruments the shader code to recognize accesses by such pointers and validate that all reads and writes are within the address ranges specified in the input buffer. Note that the instrumentation code uses 64 bit integers, and the shaderInt64 feature must be available or this validation will not be enabled.

Buffer Accesses Out of Bounds

In December 2020, GPU-assisted validation code was updated to include validation for buffer accesses beyond the declared size of the buffer. The implementation uses the input buffer described in the descriptor indexing section above to tell the instrumented shader how big a given buffer is, and the instrumentation returns an error for any accesses beyond that length. This validation is active whenever GPU-assisted validation is active and validates all shader accesses to all uniform and storage buffers.

The Motivation for GPU-Assisted Validation

Descriptor usage problems can be very hard to diagnose and debug. An OOB access or referencing an uninitialized descriptor can result in a subtle rendering error or a lost device situation where it becomes difficult to analyze the failing state. The developer may have to resort to a long series of iterative ad-hoc efforts such as modifying shaders to paint specific colors to communicate relevant parts of the shader's state. To avoid this inconvenience, a way to detect and report descriptor usage errors is needed.

Activating GPU-Assisted Validation

The layer portion of the implementation can also adversely affect application operation because of additional memory and descriptor usage. These impacts are discussed in more detail later.

Here are the options related to activating GPU-Assisted Validation:

1. Enable GPU-Assisted Validation - GPU-Assisted Validation is off by default and must be enabled.

2. Reserve a Descriptor Set Binding Slot - Modifies the value of the `VkPhysicalDeviceLimits::maxBoundDescriptorSets` property to return a value one less than the actual device's value to "reserve" a descriptor set binding slot for use by GPU validation.

   This option is likely only of interest to applications that dynamically adjust their descriptor set bindings to adjust for the limits of the device.

Enabling and Specifying Options with a Configuration File

The existing layer configuration file mechanism can be used to enable GPU-Assisted Validation. This mechanism is described on the Vulkan SDK download site (https://vulkan.lunarg.com/doc/sdk/latest), in the "Layers Overview and Configuration" document.

To turn on GPU validation, add the following to your layer settings file, which is often named `vk_layer_settings.txt`.

```
khronos_validation.enables = VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT
```

To turn on GPU validation and request to reserve a binding slot:

```
khronos_validation.enables =
VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT,
VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT
```

Some platforms do not support the configuration of the validation layers with this configuration file. Programs running on these platforms must then use the programmatic interface.

Enabling and Specifying Options with the Programmatic Interface

The `VK_EXT_validation_features` extension can be used to enable GPU-Assisted Validation at CreateInstance time. This approach involves populating an extension data structure and adding it to a `pNext` chain.

Here is sample code illustrating how to enable it:

```
VkValidationFeatureEnableEXT enables[] =
    {VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT};
VkValidationFeaturesEXT features = {};
features.sType = VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT;
features.enabledValidationFeatureCount = 1;
features.pEnabledValidationFeatures = enables;

VkInstanceCreateInfo info = {};
info.pNext = &features;
```

Add the `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT` enum to the `enables[]` array and increase the `enabledValidationFeatureCount` to reserve a binding slot.

Adding Debug Information to Shaders

While GPU Validation will work with all shaders, it will emit better information about the source file and line number location of the error if debug information is embedded in the shaders. Many shader compilers have an option for including debug information in the shader's SPIR-V bytecode. For `glslangValidator`, the option is `-g`. For `dxc`, the option is `-fspv-debug=line`, although `-Zi` will also generate debug info in the SPIR-V along with its other effects.

Typical Results

If you enable GPU-assisted validation and use it with a shader that has an OOB indexing error, you might see a message like (formatted for readability):

```
ERROR : VALIDATION - Message Id Number: 0 | Message Id Name:
UNASSIGNED-Image descriptor index out of bounds
Index of 6 used to index descriptor array of length 6.
Command buffer (CubeDrawCommandBuf #1)(0xbc8820).
Draw Index 0.
Pipeline (Pipeline #0)(0x41).
Shader Module (Cube Fragment Shader)(0x3f).
Shader Instruction Index = 110.
Stage = Fragment.  Fragment coord (x,y) = (419.5, 254.5).
Shader validation error occurred in file:
/home/user/src/Vulkan-Tools/cube/cube.frag at line 43.
43:    uFragColor = light * texture(tex[tex_ind], texcoord.xy);
```

There is a lot of information here:

This part of the message is the validation error - what actually went wrong. In this case, the index is one greater than allowed.

```
Image descriptor index out of bounds
Index of 6 used to index descriptor array of length 6.
```

This is the name and handle of the command buffer containing the Draw call that provoked the error. The application uses the `VK_EXT_debug_utils` extension to name objects, making it easier to find the command buffer in the application. If an application does not name its objects, only the handle is displayed.

```
Command buffer (CubeDrawCommandBuf #1)(0xbc8820).
```

This is the index of the Draw command that provoked the error. In this case, it is the first Draw command in the command buffer.

```
Draw Index 0.
```

This is the name and handle of the pipeline being used at the time.

```
Pipeline (Pipeline #0)(0x41).
```

This is the shader module name and handle of the shader that provoked the error.

```
Shader Module (Cube Fragment Shader)(0x3f).
```

This is the index of the specific SPIR-V bytecode instruction that provoked the error. It can be used to locate the offending instruction in a SPIR-V disassembly to learn more about the failure based on the instruction itself. It can also help locate the approximate location in the original shader source code.

```
Shader Instruction Index = 110.
```

This is stage-specific information. It varies depending on which stage the error occurred.

```
Stage = Fragment.  Fragment coord (x,y) = (419.5, 254.5).
```

If the shader is compiled with debug information, this additional source code information is displayed:

```
Shader validation error occurred in file:
/home/user/src/Vulkan-Tools/cube/cube.frag at line 43.
43:    uFragColor = light * texture(tex[tex_ind], texcoord.xy);
```

If a shader is not compiled with debug information, you see instead:

```
Unable to find SPIR-V OpLine for source information.  Build shader with
debug info to get source information.
```

This is an informative message saying that no further source code information can be given with shaders that are not compiled with debug information. See directions above for adding debug information to your shaders.

The combination of all this information should make it easier for the developer to locate the problem in the application.

Performance Impacts of GPU-Assisted Validation

The performance loss due to the instrumentation of the shaders to add OOB checking is heavily dependent on the shader. If a shader consists of little else than fetching data from an indexed descriptor, the impact is fairly high. On the other hand, if the shader is doing a lot of work and a few indexed descriptor accesses, the impact is small.

The layer-side code is not very computationally complex and so shouldn't contribute much to the CPU overhead for validation. The layer does wait for the queue to idle after every submit in order to examine results from the GPU. This can reduce the CPU/GPU processing overlap and introduce more CPU/GPU idle time. The validation message generation can get expensive for large shader programs compiled with debug information, but this path is taken only when errors are detected.

For one published game, Dota 2 with Source2 engine, the performance loss when enabling GPU-assisted validation is measured to be about a 10% reduction in frame rate.

Limitations for GPU-Assisted Validation

There are several limitations that may impede the operation of GPU-Assisted Validation:

Vulkan 1.1

Vulkan 1.1 or later is required because the GPU instrumentation code uses SPIR-V 1.3 features. Vulkan 1.1 is required to ensure that SPIR-V 1.3 is available.

Descriptor Types

The current implementation works with image and texel descriptor types:

VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER

Descriptor Set Binding Limit

This is probably the most important limitation and is related to the `VkPhysicalDeviceLimits::maxBoundDescriptorSets` device limit.

When applications use all the available descriptor set binding slots, GPU-Assisted Validation cannot be performed because it needs a descriptor set to locate the memory for writing the error report record.

This problem is most likely to occur on devices that support only the minimum required value for `VkPhysicalDeviceLimits::maxBoundDescriptorSets`, which is 4. Some applications may be written to use 4 slots since this is the highest value that is guaranteed by the specification. When such an application using 4 slots runs on a device with only 4 slots, then GPU-Assisted Validation cannot be performed.

In this implementation, this condition is detected and gracefully recovered from by building the graphics pipeline with non-instrumented shaders instead of instrumented ones. An error message is displayed informing the user of the condition.

Applications don't have many options in this situation and it is anticipated that changing the application to free a slot is difficult.

Device Memory

GPU-Assisted Validation does allocate device memory for the error report buffers. This can lead to a greater chance of memory exhaustion, especially in cases where the application is trying to use all of the available memory. However, the memory usage should be fairly small compared to that of a texture-heavy application. A rough estimate is around a couple of hundred bytes per Draw, allocated from larger block allocations of device memory. This estimate may change as the implementation evolves.

If GPU-Assisted Validation device memory allocations fail, the device could become unstable because some previously-built pipelines may contain instrumented shaders. This is a condition that is nearly impossible to recover from, so the layer just prints an error message and refrains

from any further allocations or instrumentations. There is a reasonable chance to recover from these conditions, especially if the instrumentation does not write any error records.

Descriptors

This is roughly the same problem as the device memory problem mentioned above, but for descriptors. Any failure to allocate a descriptor set means that the instrumented shader code won't have a place to write error records, resulting in unpredictable device behavior.

Other Device Limits

This implementation uses additional resources that may count against the following limits, and possibly others:

- maxMemoryAllocationCount
- maxBoundDescriptorSets
- maxPerStageDescriptorStorageBuffers
- maxPerStageResources
- maxDescriptorSetStorageBuffers
- maxFragmentCombinedOutputResources

The implementation does not take steps to avoid exceeding these limits and does not update the tracking performed by other validation functions.

Additional Considerations

Preserving Debug Info When Using Custom Optimization

If you perform your own custom shader optimizations, you may wish to bracket your optimization passes with the `--propagate-line-info` and `--eliminate-redundant-line-info` options. For example, the spirv-opt options list for unrolling loops would be:

```
--propagate-line-info --loop-unroll --eliminate-redundant-line-info
```

When using the API, this involves configuring the optimizer with `CreatePropagateLineInfoPass()` and `CreateRedundantLineInfoElimPass()`.

These steps are needed to minimize loss of debug line information during SPIR-V optimization.

How GPU-Assisted Validation Works

Doing it Yourself

One way of explaining the details of GPU-assisted validation is to show how it can be done in your own application. This is indeed possible, but it is extremely tedious and disruptive if you apply it completely across your application as the validation layer does. Going through these steps as a simple example also illustrates that it is much easier just to turn on GPU-assisted validation in the layer with just a single option switch.

Using a Storage Buffer to Collect Debug Information

The application creates a small Vulkan storage buffer before submitting a command buffer. This buffer is cleared to zero by the application and is bound to the shader via a descriptor just like any other buffer. After the command buffer completes execution, the application inspects the buffer to see if the GPU wrote anything into it.

One possible format for this buffer is:

```
{
    uint32_t count;
    uint32_t data[30];
}
```

The `count` member keeps track of how many “debug records” were attempted to be written into the `data` member. The “attempted” part of this is explained shortly.

A debug record is defined to pass back whatever information is useful from the GPU to the application when a descriptor indexing error occurs. In this simple example, there are 3 values per debug record, allowing space for 10 debug records in the `data[30]` array. The meaning of these 3 values are:

Word 0: ID
Word 1: Index
Word 2: Number of elements in the descriptor array

The ID word can be any sort of unique identifier that communicates the type of error and where it occurred. The GPU-assisted validation implementation actually uses several words to express this information and much more, but this example is kept much simpler for illustration purposes. The complete debug record used by the actual implementation is shown later in this paper.

The Index is the index that the shader program used to index the descriptor array. When there is an error, this value is greater than or equal to the number of descriptors in the array. It is probably useful to report this value to the developer as it may provide a clue about how it was computed by the application.

The number of elements in the descriptor array is also useful to report for further problem isolation and debugging purposes.

The choice of 10 records is arbitrary.

Example: Manually-Instrumented Shader Program

This is a portion of a manually-instrumented fragment shader program, written in a GLSL-like syntax.

```
layout (set = 0, binding = 1) uniform sampler2D tex[6];
layout (set = 1, binding = 0) buffer debugBuffer_t
{
   uint count;
   uint data[30];
} debugBuffer;

/* … */
layout (location = 2) flat in uint tex_ind;
layout (location = 0) out vec4 uFragColor;

void main() {
/* … */
  if (tex_ind < 6) {
     uFragColor = light * texture(tex[tex_ind], texcoord.xy);
  } else {
     uint start = atomicAdd(debugBuffer.count, 3);
     if (start < 30) {
         debugBuffer.data[start + 0] = 0xBAD00BAD;
         debugBuffer.data[start + 1] = tex_ind;
         debugBuffer.data[start + 2] = 6;
     }
     uFragColor = vec4(0);
  }
}
```

In the main function, the original line of code is:

```
uFragColor = light * texture(tex[tex_ind], texcoord.xy);
```

The additional code is instrumentation that is added manually by the developer. The actual GPU-assisted validation implementation does essentially the same thing by using a SPIR-V optimizer pass to add the same functionality directly to the SPIR-V bytecode.

The instrumentation begins by bounds-checking the descriptor index and then going ahead with the original operation if the index is in bounds.

If the index is out of bounds, the instrumentation code “reserves” a debug record inside of the `data` array. The `atomicAdd` is needed because there can be many other fragment shader executions happening in parallel. This `atomicAdd` operation guarantees that only a single fragment shader instance gets ownership of a debug record within the `data` array. The return value of the `atomicAdd` function is the value of `count` before it is incremented.

Even after the array has filled up, the `atomicAdd` continues to increment the `count` member. This is a side effect of the need to obtain atomic read and update access to this variable. This just changes the meaning of the `count` member from “records actually written” to “the number of records that were attempted to be written.” This difference does not present any problem when analyzing the storage buffer contents later since the length of the storage buffer is known.

If there is space for the debug record, the instrumentation code fills in the record with the debug information decided upon earlier.

This does mean that since there can be millions of fragment shader invocations and the buffer is fairly small, many error reports are missed, but this is unlikely to be a problem since many of these errors probably have the same root cause. Even one debug record may be enough.

Instead of attempting to dereference the descriptor array with an invalid index, the instrumented code simply returns a fragment color of `vec4(0)`. This avoids performing a memory operation that could result in a lost device.

Analyzing the Storage Buffer

When the command buffer(s) containing the draw calls that trigger the instrumented shader completes, the application examines the storage buffer. If the first word, the `count` member, is still zero, then no errors occurred and nothing else needs to be done. Otherwise, the application can walk through the debug records and report the invalid indices that the instrumented shader discovered.

DIY Summary

Clearly, there is a lot of work to do if applying the above, mostly in the modification of the shaders. It is unlikely that a developer wants to maintain shaders with this type of instrumentation implemented at the source code level. Instead, the implementation of GPU-assisted validation performs this work by using a SPIR-V optimizer pass to add

instrumentation code directly to the SPIR-V bytecode to achieve the same effect as the example above. The validation layer handles the details surrounding the buffer and its descriptors.

How the Implementation Works

Instrumenting the Shaders

The previous sections explain some of the details of the shader instrumentation performed by the SPIR-V Tools optimizer. Here is how the instrumentation is applied by the layer:

When the application creates a Shader Module, the layer intercepts this API call and calls the optimizer to add the instrumentation. The code to do this looks something like:

```
using namespace spvtools;
spv_target_env target_env = SPV_ENV_VULKAN_1_1;
Optimizer optimizer(target_env);
optimizer.RegisterPass(CreateInstBindlessCheckPass(desc_set_bind_index,
                                                   unique_shader_module_id));
optimizer.RegisterPass(CreateAggressiveDCEPass());
bool pass = optimizer.Run(new_pgm.data(), new_pgm.size(), &new_pgm);
```

There are two parameters for the InstBindlessCheck pass:

desc_set_bind_index - This design involves using an entire descriptor set for the debug buffer descriptors. (Alternate designs are discussed later). The layer chooses a descriptor set binding slot and passes it to the instrumentation code via this parameter. The instrumentation code must know this descriptor binding information at instrumentation time in order to find the debug buffer.

unique_shader_module_id - This parameter provides a means for the layer to identify the shader when it provokes an OOB indexing error. The shader instrumentation includes this ID in the error report record it generates when there is an indexing error. The layer then uses this ID to inform the user which shader caused the error.

The original SPIR-V bytecode is stored in the `std::vector` named `new_pgm` before the call to `optimizer.Run` and contains the new instrumented bytecode after the call.

The dead-code elimination pass (AggressiveDCE) is run to remove any instrumentation code that may be present and the optimizer considers as unreachable.

Setting up the Layer for GPU-Assistance

The layer implementation for GPU-assisted validation differs from most of the other validation functions in that it modifies some of the API calls passing through the layer. As shown above, the layer modifies the shader code by instrumenting it. It also modifies pipeline layouts and adds additional memory and descriptors for managing the debug buffer. These basic operations are described here:

- Determine the descriptor set binding index that is eventually used to bind the descriptor set just allocated and updated. Usually, it is `VkPhysicalDeviceLimits::maxBoundDescriptorSets` minus one. For devices that have a very high or no limit on this bound, pick an index that isn't too high, but above most other device maxima such as 32.

- Turn on the device features `fragmentStoresAndAtomics` and `vertexPipelineStoresAndAtomics`, which are necessary for the GPU instrumentation to work. If buffer device address validation is needed, the shaderInt64 feature will be enabled.

- For each draw call, the layer allocates two blocks of device memory (one for input and one for output) to hold a single debug output record written by the instrumented shader code. There is a device memory manager in the layer to handle this efficiently.

  There is probably little advantage in providing a larger buffer in order to obtain more debug records. It is likely, especially for fragment shaders, that multiple errors occurring near each other have the same root cause.

  A block is allocated on a per draw basis to make it possible to associate a shader debug error record with a draw within a command buffer. This is done partly to give the user more information in the error report, namely the command buffer handle/name and the draw within that command buffer. An alternative design allocates this block on a per-device or per-queue basis and should work. However, it is not possible to identify the command buffer that causes the error if multiple command buffers are submitted at once.

- For each draw call, the layer allocates a descriptor set and updates it to point to the block of device memory just allocated. There is a descriptor set manager in the layer to handle this efficiently. Also, make an additional call down the chain to create a bind descriptor set command to bind our descriptor set at the desired index. This has the effect of binding the device memory block belonging to this draw so that the GPU instrumentation writes into this buffer for when the draw is executed. The end result is

that each draw call has its own device memory block containing GPU instrumentation error records if any occurred while executing that draw.

- When creating a ShaderModule, pass the SPIR-V bytecode to the SPIR-V optimizer to perform the instrumentation pass. Pass the desired descriptor set binding index to the optimizer via a parameter so that the instrumented code knows which descriptor to use for writing error report data to the memory block. Use the instrumented bytecode to create the ShaderModule.

- For all pipeline layouts, add our descriptor set to the layout, at the binding index determined earlier. Fill any gaps with empty descriptor sets.

  If the incoming layout already has a descriptor set placed at our desired index, the layer must not add its descriptor set to the layout, replacing the one in the incoming layout. Instead, the layer leaves the layout alone and later replaces the instrumented shaders with non-instrumented ones when the pipeline layout is later used to create a graphics pipeline. The layer issues an error message to report this condition.

- When creating a GraphicsPipeline, check to see if the pipeline is using the debug binding index. If it is, replace the instrumented shaders in the pipeline with non-instrumented ones.

- After calling QueueSubmit, perform a wait on the queue to allow the queue to finish executing. Then map and examine the device memory block for each draw that was submitted. If any debug record is found, generate a validation error message for each record found.

Collecting and Reporting Results

The last step mentioned in the setup for the layer involves waiting on the queue to finish. The layer also issues the appropriate memory barriers to ensure that the GPU's write operations make it to the host CPU domain. The layer then proceeds by mapping the debug buffer and examining the first word. If this word is still zero, there were no errors and the layer maps and examines the debug buffer associated with the next draw.

If there is an error, the layer begins analysis of the debug buffer. As in the simple example discussed before, the debug buffer looks like:

```
struct DebugOutputBuffer_t
{
   uint Data Written Length;
   uint Data[];
}
```

Record Format

The `Data` is in the form of debug records which look like:

Word 0: Record Size
Word 1: Shader ID
Word 2: Instruction Index
Word 3: Stage
<Stage-Specific Words>
<Validation-Specific Words>

The Record Size is the number of words in this record, including the Record Size.

The Shader ID is a handle that was provided by the layer when the shader was instrumented.

The Instruction Index is the instruction within the original function at which the error occurred. For bindless accesses, this will be the instruction which consumes the descriptor in question, or the instruction that consumes the OpSampledImage that consumes the descriptor.

The Stage is the integer value used in SPIR-V for each of the Graphics Execution Models:

Stage	Value
Vertex	0
TessCtrl	1
TessEval	2
Geometry	3
Fragment	4
Compute	5
RayGenerationNV	5313
IntersectionNV	5314
AnyHitNV	5315
ClosestHitNV	5316
MissNV	5317
CallableNV	5318

Stage-Specific Words

These are words that identify in which "instance" of the shader the validation error occurred. Here are words for each stage:

Stage	Word 0	Word 1	Word 2
Vertex	VertexID	InstanceID	Unused
TessCntrl	InvocationID	PrimitiveID	Unused
TessEval	PrimitiveID	TessCoord.u	TessCoord.v
Geometry	PrimitiveID	InvocationID	Unused
Fragment	FragCoord.x	FragCoord.y	Unused
Compute	GlobalInvocID.x	GlobalInvocID.y	GlobalInvocID.z
RayGenerationNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z
IntersectionNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z
AnyHitNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z
ClosestHitNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z
MissNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z
CallableNV	LaunchIdNV.x	LaunchIdNV.y	LaunchIdNV.z

"Unused" means not relevant, but still present.

Validation-Specific Words

These are words that are specific to the validation being done. For bindless validation, they are variable.

The first word is the Error Code.

Error	Word 0	Word 1
IndexOutOfBounds	Descriptor Index	Descriptor Array Length
DescriptorUninitialized	Descriptor Index	Unused
BufferDeviceAddressOOB	Out of Bounds Address	Unused

For example, the words written for an image descriptor bounds error in a fragment shader are:

Word 0: Record size (9)
Word 1: Shader ID
Word 2: Instruction Index
Word 3: Stage (4: Fragment)
Word 4: FragCoord.x
Word 5: FragCoord.y
Word 6: Error (0: IndexOutOfBounds)
Word 7: DescriptorIndex
Word 8: DescriptorArrayLength

Note that many of the fields in the above error record definition are for items that are not yet implemented.

Finding the Source Code

It is pretty straightforward for the layer to collect the above information and then format it into an error message like the one shown in the simple example earlier. But also as seen before, it is even more useful to provide some source code information if the shader is compiled with debug info.

The process starts by associating the Instruction Index from the error record with a SPIR-V OpLine instruction.

Finding the OpLine

The familiar line of code from our example:

```
uFragColor = light * texture(tex[tex_ind], texcoord.xy);
```

results in the following SPIR-V bytecode stream, part of which is shown here disassembled:

```
                Line 1 43 0
36:    7(float) Load 24(light)
46:     39(int) Load 45(tex_ind)
48:     47(ptr) AccessChain 43(tex) 46
49:          38 Load 48
53:   33(fvec4) Load 51(texcoord)
54:   52(fvec2) VectorShuffle 53 53 0 1
55:   33(fvec4) ImageSampleImplicitLod 49 54
56:   33(fvec4) VectorTimesScalar 55 36
                Store 35(uFragColor) 56
```

The Instruction Index from the error report is 110, which corresponds to the ImageSampleImplicitLod instruction. (The numbers in the left column are *not* instruction offsets). The layer scans the SPIR-V code to find the OpLine instruction that occurs just prior to the offending instruction. In this case, it is the OpLine instruction:

```
Line 1 43 0
```

The first parameter is an ID for an OpString instruction that contains a source-level file name.

The second parameter is the line number in that file.

The third parameter is the column number. This number is usually zero, but it is the starting column number of a statement when there is more than one statement on a line of source code.

The layer locates the SPIR-V OpString instruction with the ID from the first parameter and extracts the literal string containing the filename. Here is what the OpString instruction looks like. It has an ID of 1, matching the ID from the OpLine.

```
   1:             String
"/home/user/src/Vulkan-Tools/cube/cube.frag"
```

Now the layer has enough to display the file name along with the line and column number information.

If the online shader compiler is used, the shader source is in memory and there is no file name to display. But the layer still presents the line and column number.

Finding the OpSource

The OpSource instruction's third parameter is also the ID of the OpString file name, associating the source code string with the source file name. The source code itself is stored in the fourth parameter of the OpSource as a single string literal with each source code line delimited by a newline. There is an OpSourceExtension instruction to handle programs longer than the maximum length of a string literal allowed in SPIR-V. The OpSource statement along with the first few lines of code looks like:

```
         Source GLSL 430 1  "// OpModuleProcessed client vulkan100
// OpModuleProcessed target-env vulkan1.0
// OpModuleProcessed entry-point main
#line 1
/*
 * Copyright (c) 2015-2016 The Khronos Group Inc.
…
"
```

The layer again scans the SPIR-V to locate an OpSource with the same ID as the file name.

One would then think that it is sufficient to just count newlines in the OpSource string until reaching the desired line number, which is 43 in this example, but in this case, the language processor (shader compiler) inserted a few lines of metadata and inserted a "#line 1" directive. So the layer needs to account for these lines by finding what OpSource line number contains the "#line 1" directive (4) and adding that line number to the desired line number (43) to get the correct OpSource line number.

```
 1: // OpModuleProcessed client vulkan100
 2: // OpModuleProcessed target-env vulkan1.0
 3: // OpModuleProcessed entry-point main
 4: #line 1
 5: /*
 6:  * Copyright (c) 2015-2016 The Khronos Group Inc.
…
47: uFragColor = light * texture(tex[tex_ind], texcoord.xy);
```

The numbers in the left column above are the line numbers for the lines contained in the OpSource string.

But this "rebasing of line 1" is only enough to satisfy this simple single file example.

It may also be the case that the source code comes from several different files and a language preprocessor combined the files into one stream to give to the compiler. For example:

Contents of “main”:

```
1: int m4(int a) {
2:     return a * 4;
3: }
4: #include “extra”
5: int m6(int a) {
6:     return a * 6;
7: }
```

Contents of “extra”:

```
1: int m5(int a) {
2:     return a * 5;
3: }
```

Running this through a preprocessor might result in:

```
 1: #line 1 “main”
 2: int m4(int a) {
 3:     return a * 4;
 4: }
 5: #line 1 “extra”
 6: int m5(int a) {
 7:     return a * 5;
 8: }
 9: #line 5 “main”
10: int m6(int a) {
11:     return a * 6;
12: }
```

In a more general sense, the layer needs to locate the #line directive in the source that corresponds to the file indicated by the OpLine that is closest to and before the line number indicated by the OpLine. If a #line directive does not have a file name, the source is coming from an online compile as a single string or is a single file.

In the above “preprocessor” example, if the fault occurred in “main” at line 6 (`return a * 6;`), the OpLine would indicate the source file is “main” at line number 6. But the OpSource contains the preprocessed output. The layer finds the correct #line directive at line 9 in the OpSource (`#line 5 “main”`), which tells the layer that line 10 in the OpSource corresponds to line 5 of main. Since the OpLine indicated line 6 of “main”, the layer adds 1 (6 minus 5) to 10 to get to line 11 of the preprocessed output in the OpSource, which is line 6 of “main”. Therefore, line 11 of the OpSource is the correct line.

Alternative Approach

There is more than one way to implement GPU-assisted validation. Here are some discussions about an alternative approach.

Shader Instrumentation Pipeline Adaptation

One major drawback in the current approach is that a free descriptor set binding slot is required. This approach is one way to avoid this problem.

The main concept is to defer shader instrumentation until pipeline creation time. This allows the layer to analyze the pipeline layout and “find a place” for the debug descriptors.

The most straightforward way to find a place for the debug descriptors is to just add additional bindings to one of the descriptor sets. This approach has the following considerations:

- Each shader is getting “customized” to a specific pipeline. This may reduce the opportunity for an implementation to share compiled shader modules between pipelines.
- Each shader is getting recompiled (sent to the driver as if issuing a `vkCreateShaderModule` command) for each pipeline.
- The interface to the spirv-tools optimization pass that performs shader instrumentation needs to be widened to include a new binding id/position in addition to the current descriptor set index.

The above considerations may be acceptable enough to consider adding this approach as a viable way to get around the problem of needing a free descriptor set binding slot.

References

GPU-Assisted Validation Design Document

This more detailed design document is found in the KhronosGroup/Vulkan-ValidationLayers GitHub repository in the file docs/gpu_validation.md.

GPU-Assisted Validation Source Codes

- The layer sources are found in the KhronosGroup/Vulkan-ValidationLayers GitHub repository.
- The shader instrumentation sources are found in the KhronosGroup/SPIRV-Tools GitHub repository.

Acknowledgements

The author would like to thank Greg Fischer from LunarG for spearheading the overall design and approach for GPU-assisted validation and also for the implementation of the shader instrumentation in the SPIR-V optimizer. Also, thanks to Dan Ginsburg at Valve for providing workloads and feedback during the project development.

Document Change Log

02/22/19 - Revised error codes
05/02/19 - Added section on descriptor indexing validation
08/15/19 - Added information about the compatibility of line and source info generated by dxc as it would be consumed by GPU-AV
08/30/19 - Added section for buffer device address validation and updated the Stage and Stage Specific Words tables
12/10/20 - Added section for buffer access out of bounds checking