GFXReconstruct – Portable Raytracing

February 3, 2025

Fabian Schmidt
Senior Graphics Software Engineer
LunarG, Inc.

Problem Statement

Replaying a captured raytracing application proves to be more complicated than regular rasterization-workloads. This is especially true if we attempt to replay an existing raytracing-capture on a different gpu-device/vendor/driver. We refer to this as *portable raytracing*.

The main problem is that some resources are device-specific or valid only during capturing. Vulkan provides several extensions that allow hardware vendors to more flexibly support replay on their devices. These extensions provide the concept of "opaque" addresses meaning they will attempt to re-use the same addresses on replay. Examples are buffer-device-addresses, shader-group-handles, acceleration-structure addresses and micromaps.

You can query the support for opaque resources: **vulkaninfo | grep CaptureReplay**
An example output might look something like this:

```
accelerationStructureCaptureReplay                    = true
bufferDeviceAddressCaptureReplay                      = true
micromapCaptureReplay                                 = false
rayTracingPipelineShaderGroupHandleCaptureReplay      = false
```

The presence of **bufferDeviceAddressCaptureReplay** is responsible for making many things 'just work' on the same device. But when we replay on a different device we often have to account for different sizes, indices, and capabilities of existing Vulkan memory-heaps.This can be accomplished with the 'gfxrecon-replay <capture_file.gfxr> **-m rebind** flag.

Sadly, rebinding memory-allocations currently prevents us from using some of the capture/replay features mentioned above, even if they are available.

The common absence of **rayTracingPipelineShaderGroupHandleCaptureReplay** is a challenge for replaying raytracing applications. But even if it is available, the size and stride requirements for shader-binding-tables might be different.

And while **accelerationStructureCaptureReplay** can be used to request opaque addresses for acceleration-structures, there is still something missing—acceleration-structures store their internal information in attached buffers. The structure is opaque and vendor-specific as are the required sizes for the storage and scratch-buffers required to build them. While we can reuse stored acceleration-structures and their assets during a replay, this normally only works on the same device and driver.

To summarize:

For a portable replay, resources can and will be different during capture and replay. Buffer addresses and handles might directly appear in API calls like **vkCmdTraceRays** or **vkCmdBuildAccelerationStructuresKHR**, and they also appear as content of attached buffers, which could have been manipulated on gpu-timeline. This makes it generally hard for tools to intercept or sanitize input data on the cpu timeline.

Lastly, buffer-device addresses might also appear in buffers or push-constants attached to shaders using **SPV_KHR_physical_storage_buffer**. We will not cover that here, only mention that it is a very similar problem.

Since the various capture/replay features may be unavailable for various reasons, we decided a better approach was necessary.

Solution

A possible way forward starts with tracking the resources during capturing. This allows additional content to be stored in the capture file that will assist detection of items (such as buffer addresses) during replay. Then, during replay, these handles and addresses will be fixed to generate the original desired result of the captured content.

For **VkBuffer** and **VkAccelerationStructureKHR** this means providing a back and forth mapping between their Vulkan-handles and associated VkDeviceAddresses.

VkDeviceAddess (capture) -> (VkBuffer + offset) -> VkDeviceAddess (replay)

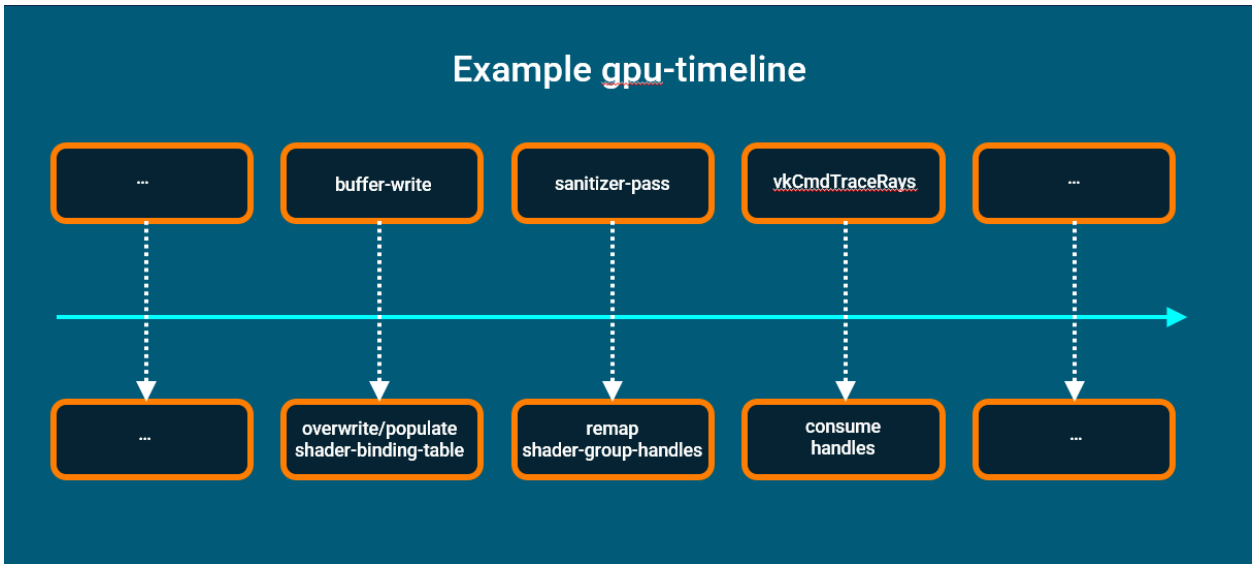
Shader-group-handles acquired with **vkGetRayTracingShaderGroupHandlesKHR** also need to remap handles at replay time using extra information gleaned during capture. We examine the **VkPhysicalDeviceRayTracingPipelinePropertiesKHR** for capture and replay, compare the sizes,

strides, and alignment requirements. This allows us to decide if the shader-binding-table's layout is compatible or a drop-in replacement will be required.

For **VkAccelerationStructureKHR** we have no choice but to recreate them using the original input-data. To do this, we must add additional tracking of associated buffers containing triangles, indices, AABB, and instance-data. The replay needs to additionally check for required buffer-sizes for storage and scratch space during builds. If the size requirements during replay are larger than during capture, we swap the original buffers with sufficiently large shadow-buffers and structures. With that in place we again need to check and potentially map from capture to replay handles.

After all this, we replace all occurrences of stale or replaced resources to avoid a gpu crash. This is straightforward for data passed directly into API-calls but more complicated for resources provided in buffers. We must wait until the data gets consumed on the gpu-timeline, otherwise any buffer copy or compute shader operation could change our sanitized buffer again.

So we know what to replace and where, leaving us with 'how'.



We inject a compute-based replacer or sanitizer pass into the existing **VkCommandBuffer** used to submit the calls to **vkCmdTraceRays** or **vkCmdBuildAccelerationStructuresKHR.** This is a simple hashmap lookup, but on gpu-timeline. We'll have a strong guarantee that we'll be last to manipulate any buffer content before submission.

Finally, all added Vulkan calls are marked as such, allowing differentiation between original behavior and portability additions.

Result

We added an initial release of portable raytracing to **gfxreconstruct** and you can find it in the upcoming Vulkan-SDK 1.4.304.1 .

Nothing changes for the default case, when replaying on the same device. The portable raytracing code paths are used only when replay is triggered using the **-m rebind** flag. Using this flag allows replay of raytracing captures taken on another gpu device or driver.

Conclusion

We can now demonstrate portable replay of raytracing applications that would previously crash during playback. This increases the usefulness and coverage of our tool for cases like bug reporting.

We believe that this general approach will also be applicable for other existing and upcoming gpu-centric extensions like **VK_EXT_device_generated_commands**.

However, there remains more work to do. For example, a future effort will apply this process to device addresses used in shaders, a.k.a. **SPV_KHR_physical_storage_buffer**