# Vulkan sparse binding - a quick overview

# Vulkan sparse binding - a quick overview

Thu
13
Dec 2018

On 13 December 2018 AMD released new version of graphics driver: 18.12.2. One could say there is nothing special about it – new drivers are released as often as every 2 weeks. But this version brings a change very important for Vulkan developers. AMD now catches up with competition – Nvidia and Intel – in supporting Vulkan **sparse binding** and **sparse residency**, which makes it usable in Windows PC games. I'd like to make it an opportunity to briefly describe what is it and how can you start using it, assuming you are a programmer who already knows a bit about C or C++ and Vulkan.

Level 0: Traditional resources

Creation of memory-consuming "resources" – buffers and images – is a 4-step process in Vulkan. You need to:

1. Create your `VkBuffer` or `VkImage`.
2. Query it for memory requirements – functions `vkGetBufferMemoryRequirements`, `vkGetImageMemoryRequirements`, structure `VkMemoryRequirements` or functions `vkGetBufferMemoryRequirements2`, `vkGetImageMemoryRequirements2`, structure `VkMemoryRequirements2`.
3. Allocate `VkDeviceMemory` object (I like to call it a "memory block"). You could make a separate allocation for each of your resources, but this doesn't scale to large projects (because there is an overhead of that, as well as limit on maximum number of allocations – `VkPhysicalDeviceLimits::maxMemoryAllocationCount`), so it is recommended to allocate larger blocks and write your own memory allocator to manage them – or use free Vulkan Memory Allocator library.
4. Bind your resource to a specific place (offset) in a memory block, by using functions `vkBindBufferMemory`, `vkImageBufferMemory` or `vkBindBufferMemory2`, `vkBindImageMemory2` and structures `VkBindBufferMemoryInfo`, `VkBindImageMemoryInfo`.

Notice these are all operations done on CPU side, effective immediately and not interacting with Vulkan queues that execute work on the GPU.

This approach has following limitations:

1. Once bound, a resource cannot be unbound or rebound to a different memory block or offset.
2. It must be bound to a single, continuous space in memory.

Because of this, after many different resources created and destroyed during runtime, memory fragmentation can occur. If you want to move data around to defragment it, you need to recreate your resources, as well as views pointing to them and update descriptors that refer to them.

Level 1: Sparse binding

Here comes "sparse binding" – a feature that can be optionally supported by a Vulkan implementation (graphics driver for a specific GPU). The main flag is `VkPhysicalDeviceFeatures::sparseBinding`. When it's set, you can bind your buffers and images in a different, more flexible way:

1. A resource is not bound to a single, continuous memory block, but rather divided into a number of equally-sized fragments (I like to call them "pages", although whether they map to real low-level memory pages is an internal implementation detail) that can be bound independently and to different memory blocks and offsets.
2. Once bound, a page can later be unbound or rebound to a different place.

Thanks to using memory allocations of same size, you can simplify your allocator and avoid fragmentation. Even when you still want to defragment (compact) your memory, you can now do so without recreating your resources – you just need to rebind them to new place where you moved their data.

One important limitation still applies: The resource must be "fully-resident" – all of its pages must be bound to a valid place in memory before it can be used on the GPU. Please also note that memory of both buffers and images is treated here as a linear sequence of pages, measured in bytes, regardless of parameters of an image like width, height, depth, or pixel format.

Here is how you can create sparse binding resources:

1. Create your image with `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` or a buffer with `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` flag set.
2. Query it for memory requirements, as above. This time though not only `VkMemoryRequirements::size` means total size of memory required for that resource and `VkMemoryRequirements::alignment` – required alignment for memory address, but the alignment

parameter also means the size of a memory page for that resource. All pages have to be of that size, and they must also be aligned to a multiply of that value.
3. Of course, you need to allocate some space in form of `VkDeviceMemory` blocks, as above.
4. In order to bind memory pages of a resource, you need to use function `vkQueueBindSparse` and structures: `VkBindSparseInfo`, `VkSparseBufferMemoryBindInfo`, `VkSparseImageOpaqueMemoryBindInfo`, `VkSparseMemoryBind`.

This time, the binding is an operation on a `VkQueue`, just like `vkQueueSubmit`. You need to use a queue that supports `VK_QUEUE_SPARSE_BINDING_BIT`. Fortunately all 3 PC GPU vendors support this flag on the main queue together with `GRAPHICS` (and AMD does on other queues as well). Vulkan spec warns that "sparse binding operations are not automatically ordered against command buffer execution, even within a single queue", which means you need to synchronize them using `VkSemaphore` (between submits on GPU queues) or `VkFence` (to wait or poll for finish on the CPU).

Level 2: Sparse residency

We can go even further. Vulkan defines, and all 3 PC GPU vendors now support, another feature on top of sparse binding, called "sparse residency". The main flags are: `VkPhysicalDeviceFeatures::sparseResidencyBuffer` – for buffers, `sparseResidencyImage2D`, `sparseResidencyImage3D`, `sparseResidency2Samples` (also for 4/8/16) – for images. When appropriate flag is set, you can create a buffer with flag `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` or an image with flag `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`. It gives you two new features:

1. Resource can now be "partially resident" – not all pages of a resource must be bound to actual memory before it is usable on the GPU.
2. Images can be bound based on extents expressed in pixels (width, height, depth) instead of linear memory pages measured in bytes.

It lets you create a large texture (known as "megatexture"), e.g. for a vast terrain, and dynamically stream its data in and out of video memory only for parts that are really needed.

Using it is much more difficult though. There are many concepts that you need to understand first. Here is an overview of symbols you need to deal with:

- Function `vkGetPhysicalDeviceSparseImageFormatProperties` and structure `VkSparseImageFormatProperties` or function `vkGetPhysicalDeviceSparseImageFormatProperties2` and structures `VkPhysicalDeviceSparseImageFormatInfo2`, `VkSparseImageFormatProperties2` – to query for memory requirements of an image in given format, especially its `imageGranularity` – size of a single page, expressed in pixels.
  - Notice the returned information are per image aspect, while multiple aspects can also be grouped together. There may also be a special `VK_IMAGE_ASPECT_METADATA_BIT` that you need to handle.
- The concept of "mip tail" – remaining mipmap levels of a texture, too small to be placed in a separate memory pages, so it is reasonable (and actually required) to bind them in a way as described in "Level 1" above. How to specifically allocate and bind the mip tail depends on flags:
  - `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`: When set, all array layers share a single mip tail region.
  - `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` (and `VkPhysicalDeviceSparseProperties::residencyAlignedMipSize`): When set, the first mip level that would contain partially used sparse blocks begins the mip tail region. Only the first N mip levels whose dimensions are an exact multiple of the sparse image block dimensions can be bound and unbound on a sparse block basis. When not set, mip levels that are as large or larger than a sparse image block in all dimensions can be bound individually.
- Function `vkGetImageSparseMemoryRequirements` and structure `VkSparseImageMemoryRequirements` or function `vkGetImageSparseMemoryRequirements2` and structures `VkImageSparseMemoryRequirementsInfo2`, `VkSparseImageMemoryRequirements2` – to query for memory requirements of a specific, given image – `VkSparseImageFormatProperties` again, but also information about mip tail.
- `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict`: When set, reads from non-resident regions return value as if it was filled with zeros. When not set, result is undefined, but it's still safe to read and write to non-bound regions – it won't crash.
- Flag `VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT`, as well as flags: `VkPhysicalDeviceSparseProperties::residencyStandard2DBlockShape`, `residencyStandard2DMultisampleBlockShape`, `residencyStandard3DBlockShape` – tell whether the binding requirements of images comply with a standard defined in Vulkan spec, or they are nonstandard and `imageGranularity` is in effect.
- When binding, you can use structures `VkSparseImageMemoryBindInfo` (this time without the "Opaque") and `VkSparseImageMemoryBind` to specify what part of the image to bind, expressed in pixels.

There is also "sparse aliasing", which allows physical memory ranges to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents. It is supported when `VkPhysicalDeviceFeatures::sparseResidencyAliased` flag is set. You can then create your buffers with `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` and images with `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flag and make use of this feature. This doesn't work with mip tail regions though.

Please note that all what I’ve described here is about feature support, not the performance. Using sparse binding may give you benefit due to not experiencing memory fragmentation or not having to recreate your resources when moving data around in memory, but it may also have some overhead on its own. First, allocating and binding many small memory pages instead of whole resources at once may be time-consuming. Second, binding is a queue operation here, just like submit, so every such call, as well as synchronization between them using semaphores and fences may have significant performance overhead. Whether sparse binding or sparse residency is a good solution for performance of your game, that’s a question which is out of scope of this article.

By the way, new version 2.2.0 of Vulkan Memory Allocator library improves support for sparse binding by adding convenience functions that allocate and free multiple memory pages at once. It also adds a test for sparse binding, which may serve as an example code.

Thank you for reading that far. If you find any mistakes in this article or have any other feedback, please leave a comment below. Further details can be found in Vulkan specification chapter 31 "Sparse Resources".