Differences in memory management between Direct3D 12 and Vulkan

Since July 2017 I develop Vulkan Memory Allocator (VMA) – a C++ library that helps with memory management in games and other applications using Vulkan. But because I deal with both Vulkan and DirectX 12 in my everyday work and I recently develop a D3D12-equivalent library - D3D12 Memory Allocator, I think it's a good idea to compare them.

This is an article about a very specific topic. It may be useful to you if you are a programmer working with both graphics APIs – Direct3D 12 and Vulkan. These two APIs offer a similar set of features and performance. Both are the new generation, explicit, low-level interfaces to the modern graphics hardware (GPUs), so we could compare them back-to-back to show similarities and differences, e.g. in naming things. For example, `ID3D12CommandQueue::ExecuteCommandLists` function has Vulkan equivalent in form of `vkQueueSubmit` function. However, this article focuses on just one aspect – memory management, which means the rules and limitation of GPU memory allocation and the creation of resources – images (textures, render targets, depth-stencil surfaces etc.) and buffers (vertex buffers, index buffers, constant/uniform buffers etc.) Chapters below describe pretty much all the aspects of memory management that differ between the two APIs.

Memory types

There are various types of memory that we can place a resource in. There is separate GPU memory (video RAM) and CPU memory (system RAM), but it's more complicated than that. If we deal with CPU-integrated graphics, the memory might be unified. Some memory may be cached or uncached, etc. We need to choose the right type for a new resource.

Vulkan has a notion of memory "heaps" and memory "types" inside them – a 2-level hierarchy. The set of heaps and types may different depending on the GPU and its driver. Indeed, it differs a lot between AMD, Nvidia, Intel, and various mobile chips. You need to query for the list of heaps and types using function `vkGetPhysicalDeviceMemoryProperties` and then make a good decision which memory type out of the available ones would be the best for your resource. For example, those with `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` are local to the GPU device, so they may work faster when accessed from the shader code. Those with `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` flag are accessible to mapping from the CPU.

In D3D12 the types of memory are called "heap types". There are three of them predefined: `D3D12_HEAP_TYPE_DEFAULT`, `D3D12_HEAP_TYPE_UPLOAD`, `D3D12_HEAP_TYPE_READBACK`. There is also `D3D12_HEAP_TYPE_CUSTOM` which lets you specify more detailed requirement for a non-standard memory, but I'm not even sure if any GPU on Windows PC supports it or any D3D12 code uses it.

Resource creation

In D3D12 the easiest way to create a resource is to use function `ID3D12Device::CreateCommittedResource`. You need to pass the description of the resource to create – incl. structure `D3D12_RESOURCE_DESC`, and requirements for the memory to place it into – incl. structure `D32D12_HEAP_PROPERTIES`. Returned object is `ID3D12Resource` – a "committed resource" created with backing memory already allocated for it. The

memory is called “implicit heap” and it’s not directly accessible. The only object you need to handle is the resource itself.

If you want to allocate a larger memory block in D3D12 and place your resources in it at different offsets, you need to first create a `ID3D12Heap`. This object represents an allocated memory block. You create it by calling function `ID3D12Device::CreateHeap`. You can then create so called “placed resource” using function `ID3D12Device::CreatePlacedResource`. You need to pass the heap and the offset along with parameters of the resource to create. The resource is then created and bound to this specific place in the memory block.

In Vulkan there is only the second way available, and it’s even more complicated. You always need to allocate a memory block manually, called `VkDeviceMemory`, by using function `vkAllocateMemory`. You can then create your resource (function `vkCreateImage`, `vkCreateBuffer`). Finally, you also need to bind those two together, using a separate function call – one of `vkBindBufferMemory`, `vkBindBufferMemory2`, `vkBindImageMemory`, `vkBindImageMemory2`. After that the buffer or image is pointing to a specific memory block at specific offset and we can use it for rendering cool graphics.

Necessity to sub-allocate

Can’t we just allocate a separate memory block for each resource? Do we really need to write or use some allocator library to reserve bigger blocks and sub-allocate resources from them? In Vulkan it is necessary because there is a limit on the maximum number of allocations that we can make. It’s called `VkPhysicalDeviceLimits::maxMemoryAllocationCount`. Depending on GPU it can be as large as 4294970000 or as small as 4096. In the latter case, any large game will probably need more resources than that. That’s why we need to sub-allocate.

In D3D12 you need not do that. You can just always use `ID3D12Device::CreateCommittedResource`, as there is no formal limit on the number of resources that can be created. However, having too many memory blocks can impose some performance penalty, so the recommendation to allocate bigger blocks (like 256 MB) and sub-allocate resources from them still holds – it’s the same for both APIs.

Query for size and offset

Before making an allocation, it is essential to know how much memory will a resource need. Both APIs provide a way to query for size required by a resource. Both also have a concept of alignment, which is a number of bytes that the offset must be multiply of, e.g. 16 B, 64 KB, or maybe even 4 MB.

In D3D12 a function serving this purpose is called `ID3D12Device::GetResourceAllocationInfo`. It returns 2 64-bit numbers – size and alignment. What is worth noting is that it takes `D3D12_RESOURCE_DESC` – a structure carrying the description of a resource we want to create. Vulkan, on the other hand, requires already created resource. Functions `vkGetBufferMemoryRequirements`, `vkGetBufferMemoryRequirements2`, `vkGetImageMemoryRequirements`, `vkGetImageMemoryRequirements2` also return size and alignment, but they need a handle to an existing `VkBuffer` or `VkImage`, respectively.

Proximity requirements

Alignment of the resources placed at specific offsets in a bigger memory block is not the only requirement. There is also a limitation on whether different types of resources can be placed next to each other. In Vulkan, there is a parameter called `VkPhysicalDeviceLimits::bufferImageGranularity`. The name is quite misleading and understanding it may not be easy, but I will try to explain in the best way I can. All resources are split into two categories: 1. "linear" (as defined by the Glossary in Vulkan spec), meaning all the buffers plus images with `VK_IMAGE_TILING_LINEAR` (this flag is not frequently used), 2. "non-linear", meaning images with `VK_IMAGE_TILING_OPTIMAL` (pretty much all the images). Now if you imagine every block of `VkDeviceMemory`, starting at its local offset 0, divided into "pages" with a size of `bufferImageGranularity` bytes, then you shouldn't place resources from different categories on a single page. To ensure that, you either need to align offsets of all your resources up to `bufferImageGranularity`, or keep the resources of these two categories separate. The first option is easier, but it may waste a lot of memory, as some cards (namely old GeForces) have this limit as high as 65536. VMA library handles this automatically and does something more intelligent – it looks at the surrounding sub-allocations to find an optimal place for a new resource that won't cause a conflict.

In D3D12 there is a similar concept, but it's defined differently. We can query the hardware capability as `D3D12_FEATURE_DATA_D3D12_OPTIONS::ResourceHeapTier`. When it's `D3D12_RESOURCE_HEAP_TIER_1`, resources are divided into three categories: 1. buffers, 2. textures that are render targets or depth-stencil, 3. all other textures. We have to allocate them out of completely separate memory heaps, and the heaps have to be created with flags that allow only one category: `D3D12_HEAP_FLAG_ALLOW_ONLY_BUFFERS`, `D3D12_HEAP_FLAG_ALLOW_ONLY_RT_DS_TEXTURES`, `D3D12_HEAP_FLAG_ALLOW_ONLY_NON_RT_DS_TEXTURES`. When the capability enum is `D3D12_RESOURCE_HEAP_TIER_2`, we can mix all types of textures together, placed in one memory heap.

Dedicated allocations

Some types of resources on some GPUs benefit from having their own, dedicated memory block. For example, a driver may then enable additional compression formats, but what's happening exactly is an implementation detail unavailable to an average developer.

In Vulkan there is a concept of a "dedicated allocation" – a memory block allocated for just one resource. This is roughly equivalent of D3D12 "committed resource". This has been added first as VK_KHR_dedicated_allocation extension and then promoted to core API of the new Vulkan 1.1. To create such allocation, you need to 1. create your resource, 2. query it for required size, 3. allocate `VkDeviceMemory`, 4. bind them together – everything like before, but this time you need to attach additional structure `VkMemoryDedicatedAllocateInfo` to `pNext` list of your `VkMemoryAllocateInfo` passed to `vkAllocateMemory` function, to specify the handle of your image or buffer at the moment you make the allocation. There is also a way to query whether a resource would benefit from having a dedicated allocation. For that you need to attach structure `VkMemoryDedicatedRequirements` to `pNext` list of your `VkMemoryRequirements2` passed to `vkGetImageMemoryRequirements2` or `vkGetBufferMemoryRequirements2` function. From it you can read whether the resource `prefersDedicatedAllocation` or even `requiresDedicatedAllocation` (the last one happens only when you import or export external memory cross-API, AFAIK). See also my article: VK_KHR_dedicated_allocation unofficial manual.

In D3D12 there is no way to know which resources would benefit from being created using `CreateCommittedResource`, as opposed to `CreatePlacedResource`. Nvidia says in their DX12 Do's And Don'ts to *"Use committed resources where possible to give the driver more knowledge"*, but it may not be the optimal strategy on GPUs from other vendors.

Query for budget

Various types of objects occupy the memory – some of them explicit, allocated with a certain size (textures, buffers), some of them implicit, with an unknown size (swap chain, command lists, pipeline objects with shaders, descriptors, queries etc.). Thus, it would be useful to know how much memory is free to use for new resources. Surprisingly, something so obvious as this was not accessible from the beginning in both APIs.

D3D12 (or more specifically – DXGI) was first to add a query for "memory budget", in form of function `IDXGIAdapter3::QueryVideoMemoryInfo` and callback `IDXGIAdapter3::RegisterVideoMemoryBudgetChangeNotificationEvent`. Vulkan added equivalent extension VK_EXT_memory_budget just recently, and it still doesn't seem to be supported by all PC GPUs (Intel chips are missing from this list at the moment of writing).

What happens when you try to allocate more memory than it is physically available depends on the driver. The allocation may fail (so you need to handle that gracefully) or it may succeed and the operating system will then migrate some other memory blocks from video memory to system memory in the background. Then everything will still work, just slower, because transfers will happen over PCIe bus. As you can see, even D3D12 and Vulkan are not low-level enough to give you real, explicit control over the memory. The best thing you can do is to stick to the queried budget or, if that is not known, to just query for the whole memory capacity and use only as much as 80% of it and hope it will be good heuristics not to oversubscribe it.

Residency management

If memory blocks can be moved between video memory and system memory, the question is whether we can manage that manually? In D3D12 we can – there are functions `ID3D12Device::Evict` and `MakeResident` that let you do that for your resources. Unlike resources migrated automatically by the system, those evicted manually cannot be used for rendering until you make them resident again. There is the whole library D3D12Residency from Microsoft that emulates convenient memory management like in D3D11 by using this functionality. In Vulkan there is no equivalent to this.

Mapping

"Mapping" is an operation that returns raw CPU pointer to the memory, so you can write or read it from your C++ (or other CPU programming language) code. Both APIs provide similar functionality, but they differ in rules and limitations defined for it.

In Vulkan, mapping, in form of `vkMapMemory` function, happens on the level of `VkDeviceMemory` block. You specify offset and size of the region to map and receive a pointer. You don't need to "unmap" before using the memory in Vulkan operations – you can leave it persistently mapped. You may only need to "flush" or "invalidate" it, which is I describe in the next chapter. However, mapping same memory region multiple times is illegal, so you have to be careful if you place multiple resources in the same memory block and you plan to

map them. That's why VMA always has to map entire memory block, just in case the user wants to have some other resource from the same block mapped.

In D3D12, mapping happens on the level of particular resources. `ID3D12Resource::Map` function is designated for this. API documentation also allows leaving a resource persistently mapped in some cases, subject to rules and limitations described in ID3D12Resource::Map method. It is allowed to map the same resource multiple times (reference counted internally) and the mapping is thread-safe. As a result, you need not care whether your resources are created as committed (with separate implicit heaps) or placed (sharing the same heap) when mapping.

Cache flush and invalidation

Vulkan has a notion of manual cache control in form of two functions: `vkFlushMappedMemoryRanges` (which you need to call after writing to a mapped memory, before it can be used in Vulkan) and `vkInvalidateMappedMemoryRanges` (which you need to call before reading from a mapped memory). But these are required only on memory types that have `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and don't have `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` flag. It occurs on mobile GPUs. While we can never be sure about the future, at the moment all 3 PC GPU vendors provide all their `HOST_VISIBLE` memory types also as `HOST_COHERENT`, so you don't really need to "flush" or "invalidate" when coding in Vulkan for PC only. Probably that's why D3D12 doesn't have such concept at all. Cache management happens automatically there, e.g. during call to Unmap.

Aliasing

By binding resources to a specified memory block at a specified offset, the new APIs allow something that was not possible in old APIs like D3D11 – memory aliasing. Multiple resources can be bound to the same or overlapping memory region. Of course, using one causes the corruption of the content of the other, so this optimization needs to be used carefully. For example, if there is a render target texture used only temporarily between two passes in a render frame, the same memory can be reused for a different texture needed for another, disjoint period during the frame. The API must be informed about this, so it knows that the content of the memory – from the perspective of the particular resource – is garbage, should be discarded or reinitialized.

In Vulkan you need to issue normal image memory barrier, with `VkImageMemoryBarrier::oldLayout = VK_IMAGE_LAYOUT_UNDEFINED`. In D3D12, there is a separate type of barrier for that, called "aliasing barrier" – use `D3D12_RESOURCE_BARRIER_TYPE_ALIASING` and fill the members of `D3D12_RESOURCE_BARRIER::Aliasing`.

Reserved resources

Finally, there is a third type of resource that can be created in D3D12, called "reserved resource" – see function `ID3D12Device::CreateReservedResource`. Also known as tiled resource, it may let you, for example, to create a very large texture and only bind some fragments ("tiles") of it to the actual memory. Vulkan provides equivalent functionality under the name "sparse binding" and "sparse residency". I've even written a separate article about it: Vulkan sparse binding - a quick overview. It sounds useful, but people say that in practice it works prohibitively slow on Windows PC, so I wouldn't recommend using it.