20180215 - Descriptor Pool Challenges in Vulkan

Vulkan Mechanics

vkCreateDescriptorPool()

> VkDescriptorPoolCreateInfo.maxSets -> enables driver to pre-allocate block of CPU memory for pool
> VkDescriptorPoolCreateInfo.pPoolSizes -> enables driver to pre-allocate block of GPU memory for pool
> **Not** using VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT -> ables driver to use linear allocators for pool

vkAllocateDescriptorSets()

> For a given VkDescriptorPool, only one CPU thread can all into this function at a time
> *And this is a core problem many developers have with Vulkan*

vkResetDescriptorPool()

> The call that frees all the sets in a pool and zeros the CPU and GPU linear allocators.

Diving into the Problem

AMD's EPYC CPU supports 32 cores and 64 threads, so going to continue in the context of attempting to support upwards of 64 CPU threads generating command buffers in parallel.

Games doing multi-core friendly job systems have a common problem: mapping of job to thread can be non-deterministic (CPU scheduling is non-deterministic). If a game attempts to use per-thread resources to avoid synchonization primitives required to handle contention with a shared resource, *non-deterministic job-to-thread mapping presents a problem: per-thread reasources need to be sized for worst case.* Vulkan API presents another problem: pools are fixed size after allocation, which means the pool needs to be conservatively sized to worst case. And finally for the usage case of dynamic descriptor sets, the engine needs at least a second set of pool(s) to double buffer.

Command buffer recording can have highly variable amount of descriptor set usage.

> Shadow rendering can be large amounts of draws with very few descriptor sets used.
> Forward shading can have large amounts of draw with large numbers of sets which could either be dynamic or static.
> Post processing passes might have a new set per draw each frame.
> Etc.

Worst case variability would be one thread ending up getting say all of forward shading with dynamic descriptor sets. That would imply to avoid needing thread synchronization primitives, pools would need to be sized to support say almost all sets per frame. Now if the title had 64 of those pools (ie running on EPYC with 64 threads), because in a given frame any CPU thread might eventually be tasked with the dynamic set forward shading, hopefully the problem is obvious: *thats 64x the memory required compared to a single core CPU.*

Workarounds?

> Base case: 64 pools/frame, each one associated with a CPU thread: *has the worst case memory problem.*
> Could have 1 pool/frame and mutex around vkAllocateDescriptorSets(). Best case for memory, but 64-way maximum contention.
> Could have 4 pools/frame and mutex around each pool. This has 64/4=16-way maximum contention.
> Could have a pool/set, requires no mutex but each set carries overhead of a full VkDescriptorPool.
> Etc.

Objectively speaking no available option is great for memory, performance or complexity.

Taking a Broader View

The Vulkan API went to great lengths to support the "linear allocator" model (ie allocation is a simple ADD operation) for pools so that drivers could optimize for the common "per-frame-reset" usage model.

One could argue that this was designed to enable simple lock-free allocation: just change the ADD to an ATOMIC ADD. With 64-bit atomics one could even pack the {32-bit GPU, 32-bit CPU} offsets for the pool's linear allocator into one atomic operation. But perhaps somehow in the process of Vulkan's API design, that the standards body lost sight of how the functions would commonly be used in game engines in context of multi-core machines.

Cost for a non-contended atomic add (LOCK XADD) according to Agner Fog Instruction Tables for Ryzen is ~23 clocks. Using that number, base estimated non-contention ADD to ATOMIC ADD change cost for allocating even an insane number of 10000 sets/frame at 60 Hz on a 64 thread 2 GHz EPYC is roughly 0.01%. Meaning likely not practically even measurable given standard variability of CPU work.

API elegance, if such a term could ever be used in the context of Vulkan, might just be a single flag away.

VK_DESCRIPTOR_POOL_CREATE_LOCK_FREE_DESCRIPTOR_SET_BIT_EXT

That flag does not exist, but if it did, using that flag in vkCreateDescriptorPool() ***without*** using VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT could be all that is required to possibly enable multiple threads to call vkAllocateDescriptorSets() in parallel, *which completely solves a core problem developers have with Vulkan descriptor sets.*

Diving into the AMD Vulkan Driver

Thanks to AMD open-sourcing the Vulkan driver we can dive into the source on github and check if the AMD driver uses a linear allocator or not, and if so, estimate how complex a change switching to ATOMIC ADD would be in practice.

Starting with vk_descriptor_pool.cpp, tracking down the **DescriptorPool::AllocDescriptorSets()** entry point.

The CPU allocation is done in **m_setHeap.AllocSetState(&pDescriptorSets;[allocCount])**. Following into that function, shows that the driver indeed uses a linear allocator ***pSet = m_pHandles[m_nextFreeHandle++]**.

The GPU allocation is done in **m_gpuMemHeap.AllocSetGpuMem(pLayout, &setGpuMemOffset;, &pSetAllocHandle;)**. Following into that function shows that the driver does indeed special case 'no free support'.

```
bool oneShot = (m_usage & VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT) == 0
```

Which also uses a linear allocator,

```
const Pal::gpusize gpuBaseOffset = Util::Pow2Align(m_oneShotAllocForward, alignment);
. . .
m_oneShotAllocForward = gpuBaseOffset + byteSize
```

So yeah seems plausable to easily implement by at least one vendor.