Vulkan Device Memory

Posted on July 21, 2016 by Timothy Lottes

Device Memory, driver, GCN, Heap, Memory Type, Vulkan

EDIT: 2016/08/08 – Added section on Targeting Low-Memory GPUs

This post serves as a guide on how to best use the various Memory Heaps and Memory Types exposed in Vulkan on AMD drivers, starting with some high-level tips.

- **GPU Bulk Data**
  Place GPU-side allocations in DEVICE_LOCAL without HOST_VISIBLE. Make sure to allocate the highest priority resources first like Render Targets and resources which get accessed more often. Once DEVICE_LOCAL fills up and allocations fail, have the lower priority allocations fall back to CPU-side memory if required via HOST_VISIBLE with HOST_COHERENT but without HOST_CACHED. When doing in-game reallocations (say for display resolution changes), make sure to fully free all allocations involved before attempting to make any new allocations. This can minimize the possibility that an allocation can fail to fit in the GPU-side heap.
- **CPU-to-GPU Data Flow**
  For relatively small total allocation size (under 256 MB) the DEVICE_LOCAL with HOST_VISIBLE is the perfect Memory Type for CPU upload to GPU cases: the CPU can directly write into GPU memory which the GPU can then access without reading across the PCIe bus. This is great for upload of constant data, etc.
- **GPU-to-CPU Data Flow**
  Use HOST_VISIBLE with HOST_COHERENT and HOST_CACHED. This is the only Memory Type which supports cached reads by the CPU. Great for cases like recording screen-captures, feeding back Hierarchical Z-Buffer occlusion tests, etc.

Pooling Allocations

EDIT: Great reminder from Axel Gneiting (leading Vulkan implementation in DOOM® at id Software), make sure to pool a group of resources, like textures and buffers, into a single memory allocation. On Windows® 7 for example, Vulkan memory allocations map to WDDM Allocations (the same lists seen in GPUView), and there is a relatively high cost associated for a WDDM Allocation as command buffers flow through the WDDM based driver stack. Having 256 MB per DEVICE_LOCAL allocation can be a good target, takes only 16 allocations to fill 4 GB.

Hidden Paging

When an application starts over-subscribing GPU-side memory, DEVICE_LOCAL memory allocations will fail. It is also possible that later during application execution, another application in the system increases its usage of GPU-side memory, resulting in dynamic over-subscribing of GPU-side memory. This case can result in an OS (for instance Windows® 7) to silently migrate or page GPU-side allocations to/from CPU-side as it time-slices execution of each application on the GPU. This can result in visible “hitching”. There is currently no method to directly query if the OS is migrating allocations in Vulkan. One possible workaround is for the app to detect hitching by looking at time-stamps, and then actively attempting to reduce DEVICE_LOCAL memory consumption when hitching is detected. For example, the application could manually move around resources to fully empty DEVICE_LOCAL allocations which can then be freed.

EDIT: Targeting Low-Memory GPUs

When targeting a memory surplus, using DEVICE_LOCAL+HOST_VISIBLE for CPU-write cases can bypass the need to schedule an extra copy. However in memory constrained situations it is much better to use DEVICE_LOCAL+HOST_VISIBLE as an extension to the DEVICE_LOCAL heap and use it for GPU Resources like Textures and Buffers. CPU-write cases can switch to HOST_VISIBLE+COHERENT. The number one priority for performance is keeping the high bandwidth access resources in GPU-side memory.

Memory Heap and Memory Type – Technical Details

Driver Device Memory Heaps and Memory Types can be inspected using the Vulkan Hardware Database. For Windows AMD drivers, below is a breakdown of the characteristics and best usage models for all the Memory Types. Heap and Memory Type numbering is not guaranteed by the Vulkan Spec, so make sure to work from the Property Flags directly. Also note memory sizes reported in Vulkan represent the maximum amount which is shared across applications and driver.

- **Heap 0**
  - VK_MEMORY_HEAP_DEVICE_LOCAL_BIT
  - Represents memory on the GPU device which can not be mapped into Host system memory
  - Using 256 MB per `vkAllocateMemory()` allocation is a good starting point for collections of buffers and images
  - Suggest using separate allocations for large allocations which might need to be resized (freed and reallocated) at run-time
  - **Memory Type 0**

    - VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
    - Full speed read/write/atomic by GPU
    - No ability to use `vkMapMemory()` to map into Host system address space
    - Use for standard GPU-side data
- **Heap 1**
  - VK_MEMORY_HEAP_DEVICE_LOCAL_BIT
  - Represents memory on the GPU device which can be mapped into Host system memory
  - Limited on Windows to 256 MB
    - Best to allocate at most 64 MB per `vkAllocateMemory()` allocation
    - Fall back to smaller allocations if necessary
  - **Memory Type 1**
    - VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
    - VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
    - VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
    - Full speed read/write/atomic by GPU
    - Ability to use `vkMapMemory()` to map into Host system address space
    - CPU writes are write-combined and write directly into GPU memory
      - Best to write full aligned cacheline sized chunks
    - CPU reads are uncached
      - Best to use Memory Type 3 instead for GPU write and CPU read cases
    - Use for dynamic buffer data to avoid an extra Host to Device copy
    - Use for a fall-back when Heap 0 runs out of space before resorting to Heap 2
- **Heap 2**
  - Represents memory on the Host system which can be accessed by the GPU
  - Suggest using similar allocation size strategy as Heap 0
  - Ability to use `vkMapMemory()`
  - GPU reads for textures and buffers are cached in GPU L2
    - GPU L2 misses read across the PCIe bus to Host system memory
    - Higher latency and lower throughput on an L2 miss
  - GPU reads for index buffers are cached in GPU L2 in Tonga and later GPUs like FuryX
  - **Memory Type 2**
    - VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
    - VK_MEMORY_PROPERTY_HOST_COHERENT_BIT

    - CPU writes are write-combined
    - CPU reads are uncached
    - Use for staging for upload to GPU device
    - Can use as a fall-back when GPU device runs out of memory in Heap 0 and Heap 1
- **Memory Type 3**
    - VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
    - VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
    - VK_MEMORY_PROPERTY_HOST_CACHED_BIT
    - CPU reads and writes go through CPU cache hierarchy
    - GPU reads snoop CPU cache
    - Use for staging for download from GPU device

Choosing the correct Memory Heap and Memory Type is a critical task in optimization. A GPU like Radeon™ Fury X for instance has 512 GB/s of DEVICE_LOCAL bandwidth (sum of any ratio of read and write) but the PCIe bus supports at most 16 GB/s read and at most 16 GB/s write for a sum of 32 GB/s in both directions.

Timothy Lottes is a member of the Developer Technology Group at AMD. Links to third party sites are provided for convenience and unless explicitly stated, AMD is not responsible for the contents of such linked sites and no endorsement is implied.