DXVK: Binary Semaphores for CmdLists

Lee Gao - May 18, 2026

Background

See https://github.com/doitsujin/dxvk/issues/4484

In https://github.com/doitsujin/dxvk/pull/4380, the inter-queue and inter-cmdlist synchronization approach was changed to rely on timeline semaphores instead of the existing interlocked binary semaphores design.

Unfortunately, on Turnip, it appears that the Mesa Vulkan common runtime's emulated implementation of timeline semaphores on top of binary semaphores+fences contains additional performance overhead, and putting these emulated timeline semaphores in the critical path has created significant performance regressions of up to 30%+ drop in FPS in certain games tested by the community. DXVK does not scope in Android/mobile runtime environments within its purview, and are thus reluctant to provide an official fix.

This design attempts to bring back the existing binary semaphore synchronization for dxvk_cmdlist to DXVK 2.7.1+. Future work is required to understand and design a principled fix for this problem (by either making Turnip's timeline semaphores more performant or else by MitM-ing the emulated semaphores within dxvk and replacing them with more efficient synchronization primitives).

D3D Command Lists, VK Command Buffers, and Queues

When d3d11 command lists are translated into Vulkan, dxvk internally tracks them within dxvk_cmdlist objects. When a d3d command list is submitted, its corresponding DxvkCommandList::submit is called to actually populate the various vulkan command buffers, and then optimally schedule them onto the available vulkan queues.

In particular, dxvk uses 3 distinct queue families offered by Vulkan during its translation efforts:

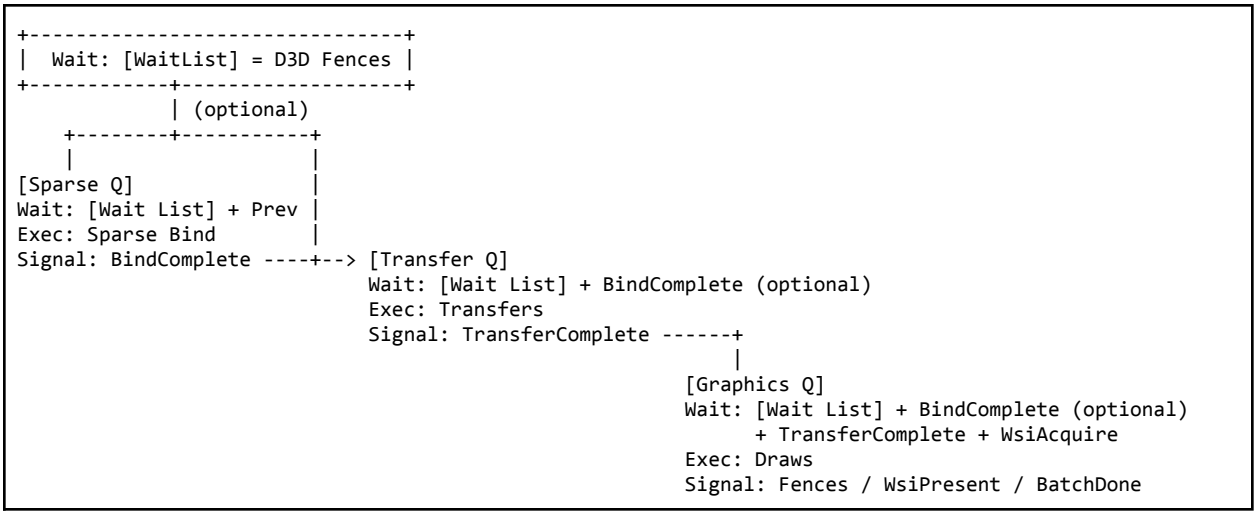
1. Graphics queue - the default draw/compute queue used to push all typical commands
2. Transfer queue - dxvk will opportunistically pull forward transfer operations and schedule them onto the dedicated transfer queue if this queue family is available on the device.
3. Sparse queue - when available, sparse bindings are processed by this dedicated queue.

In practice, most (all?) Adreno devices lack the silicon for a dedicated transfer queue, and a quick peek at vulkaninfo for new Adreno 830 devices shows only support for the default gfx/compute/transfer queue family as well as the sparse binding queue family.
The d3d command lists are often broken down into batches/blocks of commands.

During translation, each d3d command list undergoes the following steps under dxvk:
(*note*: the synchronization operations are intentionally abstracted away into happens-before/after relationships rather than concrete semaphore signals/waits)

- **Wait** for any application (D3D11) fences (DxvkFences). (src)
    - These are translated themselves into Vulkan timeline semaphores, and dxvk's CommandSubmission configuration object provides an API to wait on and signal these and generic timeline semaphores at specific timeline values:
      ```
      waitSemaphore(ts, ts_value, pipeline_stage);
      signalSemaphore(ts, ts_value, pipeline_stage);
      ```
- If this command list is batched into several blocks: (src)
    - **Wait** for all existing work to complete on the graphics (draw) queue, which are previous batches/blocks of draws issued from the same command list
- If sparse bindings are requested: (src)
    - **Wait** for all existing work to complete:
        - d3d application fences signaled
        - previous batches/blocks of draws completed (if any)
    - **Execute** the sparse bind commands on the sparse queue
    - Once the GPU completes the sparse bindings:
        - **Signal** that the sparse bindings are completed
- If dedicated transfer queue exists: (src)
    - **Wait** for all existing work to complete:
        - d3d application fences signaled
        - previous batches/blocks of draws completed (if any)
        - sparse bindings are completed (if any)
    - **Execute** the transfer commands on the transfer queue
    - Once the GPU completes the transfers:
        - **Signal** that the transfer commands are completed
- Submit the actual draw calls for this batch/frame: (src)
    - **Wait** for all existing work to complete:
        - d3d application fences signaled
        - previous batches/blocks of draws completed (if any)
        - sparse bindings are completed (if any)
        - transfers completed (if any)
    - **Wait** for swapbuffer->acquire (if first batch)
    - **Execute** the draw commands on the graphics queue
    - Once the GPU completes the draws:
        - **Signal** all d3d application fences waiting on this frame/batch
        - **Signal** that swapbuffer->present is safe to continue (if final batch)
        - **Signal** that this batch/block of draws has completed

Each group/family of commands cascade against each other. Sparse binds wait for the previous draws to complete. Transfers wait for the current sparse binds to complete. Draws wait for the current transfers to complete. So on and so on.



Now, the actual synchronization primitives used to enforce these happens-before inter-queue/inter-commandlist synchronization was changed in https://github.com/doitsujin/dxvk/pull/4380, from cascaded *binary semaphores* that explicitly encode the relationship into more implicitly defined monotone *timelines*.

The Status Quo (Binary Semaphore Approach)

Before https://github.com/doitsujin/dxvk/pull/4380, dxvk still used timeline semaphores to model DxvkFences as part of their gradual migration away from Vulkan 1.0 era synchronization primitives. However, the migration was incomplete before this PR.

The concrete implementations of Wai/Signal are done using a combination of binary VkSemaphores (for inter-queue/inter-cmd-buffer synchronization), VkFences (for cpu-side waits), and pipeline barriers (which can effectively add a VkSemaphore to the signal-list of already submitted commands on a queue, thanks to the use of `VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT`)

Sparse Binds

```
if (sparseBind) {
  // Goal: wait for the previous gfx queue to complete work
  // First, submit an empty command buffer that signals the bind sem
  // which will execute after all commands complete (bottom-of-pipe)
```

```
        m_commandSubmission.signalSemaphore(m_bindSemaphore, 0,
VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);
        if ((status = m_commandSubmission.submit(m_device, graphics.queueHandle)))
          return status;

        // Next, the sparse binds must wait for the bind sem to be complete,
        // and will signal the post sem for the next queue
        sparseBind->waitSemaphore(m_bindSemaphore, 0);
        sparseBind->signalSemaphore(m_postSemaphore, 0);
        if ((status = sparseBind->submit(m_device, sparse.queueHandle)))
          return status;

        // Finally, the next submission must wait for the post sem before
        // starting execution of the first command (top-of-pipe)
        m_commandSubmission.waitSemaphore(m_postSemaphore, 0,
VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT);
      }
```

Transfers

```
      if (m_device->hasDedicatedTransferQueue() && !commandSubmission.isEmpty()) {
        // The transfer queue's commands will signal the sdma sem for the next
        // queue
        m_commandSubmission.signalSemaphore(m_sdmaSemaphore, 0,
VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);
        if ((status = m_commandSubmission.submit(m_device, transfer.queueHandle)))
          return status;

        // The next queue must wait for the sdma sem before starting exec
        m_commandSubmission.waitSemaphore(m_sdmaSemaphore, 0,
VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT);
      }
```

Draws

```
// Waits on d3d fences, and either post-bind or sdma, configured previously
m_commandSubmission.signalFence(m_fence);
```

Note that the draw->draw synchronization is taken care of by the DxvkFence translations, so there's no explicit wait for any internal semaphores here. The CPU-side fence is used in the cleanup thread to decide when resources are safe to destroy.

The Change (Timeline Semaphore Approach)

This cascade inter-queue locking can be easily expressed in terms of a set of monotone timeline sync-points.

Sparse Binds

```
      if (sparseBind) {
        // Goal: wait for the previous gfx queue to complete work

        // First, submit an empty command buffer that signals the bind sem
        // which will execute after all commands complete (bottom of pipe)
        m_commandSubmission.signalSemaphore(m_bindSemaphore, 0,
VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);
        if ((status = m_commandSubmission.submit(m_device, graphics.queueHandle)))
          return status;

        // We can just wait for the direct timeline sync points that denote
        // all previous work on gfx+tx to be done
        sparseBind->waitSemaphore(semaphores.graphics, timelines.graphics);
        sparseBind->waitSemaphore(semaphores.transfer, timelines.transfer);

        // Reuse the graphics timeline, once sparse bindings are done, future
        // consumers must wait for the next graphics timeline point (G+1_
        sparseBind->signalSemaphore(semaphores.graphics, ++timelines.graphics);
        if ((status = sparseBind->submit(m_device, sparse.queueHandle)))
          return status;

        // Finally, the next submission must wait for the sparse work to be done
        // by waiting at gfx=G+1
        m_commandSubmission.waitSemaphore(semaphores.graphics,
          timelines.graphics, VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT);
      }
```

Transfers

```
      if (m_device->hasDedicatedTransferQueue() && !commandSubmission.isEmpty()) {
        m_commandSubmission.signalSemaphore(m_sdmaSemaphore, 0,
VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);

        // The transfer queue's commands will signal the next TX timeline point
        m_commandSubmission.signalSemaphore(semaphores.transfer,
          ++timelines.transfer, VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);
        if ((status = m_commandSubmission.submit(m_device, transfer.queueHandle)))
          return status;

        m_commandSubmission.waitSemaphore(m_sdmaSemaphore, 0,
VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT);

        // The next queue must wait for the transfer work to be done by waiting
        // at the new tx=T+1 timeline point
        m_commandSubmission.waitSemaphore(semaphores.transfer,
          timelines.transfer, VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT);
      }
```

Draws

```
      m_commandSubmission.signalFence(m_fence);

      // Once draws are done, signal the next gfx timeline point
      m_commandSubmission.signalSemaphore(semaphores.graphics,
        ++timelines.graphics, VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);

      // If there are WSI semaphores involved, do another submit only
      // containing a timeline semaphore signal so that we can be sure
      // that they are safe to use afterwards.
      if ((m_wsiSemaphores.present || m_wsiSemaphores.acquire) && isLast) {
        m_commandSubmission.signalSemaphore(semaphores.graphics,
          ++timelines.graphics, VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT);
        if ((status = m_commandSubmission.submit(m_device,
graphics.queueHandle)))
          return status;
      }
```

Turnip Timeline Semaphore Emulation

WIP (not really needed for this design, still needs more research here)

On KGSL, Turnip does not have natively supported timeline semaphores. Instead, it emulates this feature using vk_sync objects (from the Mesa Vulkan CRT) that support basic binary semaphore semantics. These primitives are enough to build the timeline_sync_point interface used by the Mesa Vulkan CRT to implement a non-hardware-supported timeline semaphore.

More details to follow, there may be some useful nuggets in the code to explain the slowdown. What's interesting is that just using a TS emulation layer on my local AMD machine, I *don't* see the same slowdown when moving to dxvk 2.5.

See https://github.com/mirror/mesa/blob/main/src/vulkan/runtime/vk_sync_timeline.c

Design

Sketch:

- Add an environment variable (DXVK_DISABLE_TIMELINE_SEMAPHORES) to optionally restore the binary semaphore inter-queue synchronization.
- Reuse DxvkTimelineSemaphores to track the fallback binary semaphores and the cpu fence
- Use an object pool for DxvkTimelineSemaphores (binary semaphores are consumed on use, so they trivially self-reset) to avoid creating/destroying too many semaphores and fences unnecessarily

- Update DxvkCommandSubmission::Submit to also track VkFences again
- Add fallback branches to revert to fallback binary semaphores when use_fences is true, otherwise keep the original code untouched.

The major risk to this approach is the need to rebase and merge this whenever dxvk updates.

See https://github.com/leegao/dxvk/commit/e950b4a90359acebf13f5a76cfa2cfaa7c238b74

Alternatives

- [Infeasible] Improve Turnip's timeline sync_data performance to reduce the TS emulation overhead
- Use VK layer to target + intercept dxvk's use of TS for queue
- Convince dxvk to adopt an official binary-semaphore fallback