Vulkan Synchronization Made Easy (without rendergraphs)

Grigory Dzhavadyan, Independent
Context
Designing a more approachable API on top of Vulkan

http://nice.graphics

- Graphics API abstraction layer (RHI) with back-ends for Metal and Vulkan.
- Designed to be at the “middle” level of abstraction.
- Sync in Vulkan backend was a serious pain point.
Result
Screenshot Courtesy of Triada Studio

Catoo
Result

Screenshot Courtesy of Triada Studio
Basic Case
Single command buffer

Image A

Buffer B

Render pass writes to A

Compute pass writes to B

Render pass samples from A reads from B
Basic Case
Single command buffer

Render pass writes to A — Compute pass writes to B — Render pass samples from A reads from B

“Trivial” to deduce the necessary barriers
Not So Basic Case
Multiple command buffers

• Order of submission is not known a priori.
• Therefore, can’t emit the correct memory barriers as commands are recorded.
• This problem arises with both multi- and single-threaded recording.
Solution: Interim Barriers
Part 1: independently track resources per command buffer

- Each resource has:
  - a single *global* synchronization state
  - one *local* synchronization state per each command buffer it is used in.
- Assume the *first* access of a pipeline stage to a given resource in a command buffer will NOT need synchronization.
- Remember the first accesses of each pipeline stage that touches the resource. Collectively, those are the “expected sync state”.
- Track subsequent accesses of a pipeline stage to the resource using the resource’s local synchronization state.
Solution: Interim Barriers
Part 2: insert barriers between command buffers

- All command buffers are submitted on a single thread, forming an ordered timeline.

- Infer the necessary barriers by comparing the *expected* sync states of the resources participating in the upcoming command buffer to their corresponding *current* global sync states.

- Record and submit the inferred barriers in an auxiliary command buffer before submitting the upcoming “main” command buffer.

- Update the current global states of all the participating resources according to the last known local sync state.

---

**Image A**

<table>
<thead>
<tr>
<th>Command Buffer X</th>
<th>Command Buffer Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>Expected sync state</th>
<th>Local sync state</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Expected sync state</td>
<td>Local sync state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>Expected sync state</th>
<th>Local sync state</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Expected sync state</td>
<td>Local sync state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>Expected sync state</th>
<th>Local sync state</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Expected sync state</td>
<td>Local sync state</td>
</tr>
<tr>
<td>C</td>
<td>Expected sync state</td>
<td>Local sync state</td>
</tr>
</tbody>
</table>

---
Hazard Tracking Inside Command Buffers

Synchronization requests

- Sync requests describe an operation we intend to perform on a resource.
- Issued when we’re about to record commands that may result in resource memory being read or modified.
- Might or might not result in a barrier, depending on the resource’s sync state.

```c
// Stage + access masks
typedef struct ngfvk_sync_barrier_masks {
    VkAccessFlags access_mask;
    VkPipelineStageFlags stage_mask;
} ngfvk_sync_barrier_masks;

// Specifies the intent to access a resource.
typedef struct ngfvk_sync_req {
    ngfvk_sync_barrier_masks barrier_masks;
    VkImageLayout layout;
} ngfvk_sync_req;
```
Hazard Tracking Inside Command Buffers
Rules for handling sync requests

- Concurrent reads are (almost) always allowed
- Only a single pipeline stage can be modifying the resource at a time.
- Once an access within a stage has “seen” the preceding write, it needs no further synchronization until the resource is modified again.
- Layout transitions need to be treated as writes.
Hazard Tracking Inside Command Buffers

Command buffer resource table

• A flat hash table keyed by a 64-bit resource handle, using open addressing.
• Memory reused between frames.
• We have precise control of re-hashing policy.
Hazard Tracking Inside Command Buffers
Resource table entry

typedef struct ngfvk_sync_res_data {
    // Expected sync state.
    ngfvk_sync_req expected_sync_state;
    // Latest synchronization state.
    ngfvk_sync_state local_sync_state;
    //...
} ngfvk_sync_res_data;
Hazard Tracking Inside Command Buffers

Resource synchronization state

// Synchronization state
typedef struct ngfvk_sync_state {
    // What access in what stage has modified the resource last.
    ngfvk_sync_barrier_masks last_writer;

    // Which accesses in which stages have seen the last write.
    uint32_t per_stage_readers;

    // …

    // Current layout (images only).
    VkImageLayout layout;
} ngfvk_sync_state;
Hazard Tracking Inside Command Buffers
What happens when a pipeline stage needs to access a resource?

• Issue a synchronization request for the access needed by the pipeline stage against the current local synchronization state.

• Keep in mind that local sync state starts out as “blank slate”: no writers, no readers.

• If no barriers have been generated for this resource up until this point, update the expected synchronization state.
Hazard Tracking Inside Command Buffers
Deciding when to emit barriers

- If a pipeline stage is requesting non-modifying access:
  - Has there been a preceding write?
    - No: just update the corresponding access bits in the per stage readers mask. No barrier emitted.
    - Yes:
      - Has this access in this stage already seen the effects of the preceding write?
        - Yes: no-op
        - No: emit barrier, update the corresponding access bits in the per stage readers mask.

// Synchronization state
typedef struct ngfvk_sync_state {
  // What access in what stage has modified the resource last.
  ngfvk_sync_barrier_masks last_writer;
  // Which accesses in which stages have seen the last write.
  uint32_t per_stage_readers;
  // Current layout (images only).
  VkImageLayout layout;
} ngfvk_sync_state;
Hazard Tracking Inside Command Buffers
Deciding when to emit barriers

• If a pipeline stage is requesting modifying access:
  • Sync with preceding reads/writes (if there are any).
  • Update the last writer.
  • Zero out per stage readers mask.
  • Update current layout.

• A non-modifying access that requires a layout transition is a bit of a special case, need to add the stage/access to per stage readers mask immediately.

// Synchronization state
typedef struct ngfvk_sync_state {
  // What access in what stage has modified the resource last.
  ngfvk_sync_barrier_masks last_writer;

  // Which accesses in which stages have seen the last write.
  uint32_t per_stage_readers;

  // Current layout (images only).
  VkImageLayout layout;
} ngfvk_sync_state;
Hazard Tracking Inside Command Buffers

Coalescing barriers

• Pending sync requests are handled in bulk (minimize vkCmdPipelineBarrier calls).

• For compute, handle them just before the dispatch.

• For graphics, handle at the end of the render pass

  • Barrier synchronization scopes are limited to subpass for barriers emitted inside the render pass. We have to emit all the necessary barriers _before_ actually recording the render pass commands.

  • VK_KHR_dynamic_rendering fixes this.

• Use sync2 wherever possible
Hazard Tracking Across Command Buffers

Emitting interim barriers

- All cmd buffers are submitted from the same single thread; it is the only thread that touches resources’ global sync states.

- Sync request generation using the *expected* access/layout from the upcoming command buffer’s resource table, targeting the global sync state.

- Update the resource’s global sync state according to the *final* local sync state from the upcoming command buffer.

- Any barriers generated are coalesced and written to an auxiliary cmd buffer which is submitted before the upcoming main cmd buffer.
Limitations
(Of this particular implementation)

• Single queue only.
  • Theoretically extensible to a multi-queue model. Maybe, someday.
  • Probably want to address it for async compute…

• No resource aliasing.
  • nicegraf does not expose memory allocation so that’s not relevant for us.

• No stores/atomics in vert/frag shaders.

• Poor sync granularity.
  • Could track individual mip levels or predefined disjoint buffer regions.

• Sync commands are issued exactly at the point they’re required, limiting implementation’s ability to overlap e.g. layout transitions with other work.
Future
Can we have VK_LAYER_KHRONOS_synchronization please?!

• VMA has solved memory management:
  • Pretty much industry standard
  • Still possible to have finer grained control (and don’t have to choose either/or)
• Why not repeat the same success story for synchronization?
**Thanks!**

<table>
<thead>
<tr>
<th>Q &amp; A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Twitter</strong></td>
<td><a href="http://twitter.com/nice_byte">http://twitter.com/nice_byte</a></td>
</tr>
<tr>
<td><strong>Mastodon</strong></td>
<td><a href="http://mastodon.gamedev.place/@nicebyte">http://mastodon.gamedev.place/@nicebyte</a></td>
</tr>
</tbody>
</table>