Vulkan Timeline Semaphores

January 15, 2020 vulkan (/news/tags/tag/vulkan), tutorial (/news/tags/tag/tutorial)

The original Vulkan synchronization APIs relied on two separate coarse-grained primitives: VkSemaphore and VkFence. Both of these were reusable binary-state objects with slightly different purposes and behavior. VkSemaphore allowed applications to synchronize operations across device queues. VkFence facilitated device to host synchronization. Together, they enabled applications to observe and control the execution of command buffers and other queue commands, but they inherited various limitations of the underlying OS and device mechanisms at the time which made them somewhat difficult to use.

The new timeline semaphore synchronization API shipping as VK_KHR_timeline_semaphore, and a core feature of Vulkan 1.2, defines a primitive containing a superset of both the original VkSemaphore and VkFence primitives, while simultaneously eliminating many of the most painful limitations of the previous APIs. In brief, timeline semaphores:

- Are a synchronization primitive whose state consists of a monotonically increasing 64-bit integer value
- Enable omnidirectional synchronization between device and host using a single primitive
- Allow wait-before-signal submission order
- Allow applications to ignore signal operations in certain cases
- Eliminate the need to reset after a signal operation before reuse
- Allow multiple wait operations per signal operation

Most Vulkan queue operations can now accept either binary or timeline semaphores, though the window system integration APIs are currently a notable exception. Important caveats also apply when a mixture of binary and timeline semaphores are used within the same workflow, as we will see below.

The Timeline Semaphore API

Before delving into the benefits of these defining features, let’s take a brief look at the API changes which implement them. Timeline semaphores are an extension of the existing VkSemaphore objects, so creating them is similar to creating a regular, or “binary” VkSemaphore object:

```
VkSemaphoreTypeCreateInfo timelineCreateInfo;
timelineCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO;
timelineCreateInfo.pNext = NULL;
timelineCreateInfo.semaphoreType = VK_SEMAPHORE_TYPE_TIMELINE;
timelineCreateInfo.initialValue = 0;

VkSemaphoreCreateInfo createInfo;
createInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
createInfo.pNext = &timelineCreateInfo;
createInfo.flags = 0;

VkSemaphore timelineSemaphore;
vkCreateSemaphore(dev, &createInfo, NULL, &timelineSemaphore);
```

Note the ability to define an arbitrary initial value for the semaphore state.

Like binary semaphores, timeline semaphores can be signaled after most queue operations complete, and waited on before beginning most queue operations. An extension structure is defined which allows applications to specify the additional state associated with timeline semaphore wait and signal operations:

```
const uint64_t waitValue = 2; // Wait until semaphore value is >= 2
const uint64_t signalValue = 3; // Set semaphore value to 3

VkTimelineSemaphoreSubmitInfo timelineInfo;
timelineInfo.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
timelineInfo.pNext = NULL;
timelineInfo.waitSemaphoreValueCount = 1;
timelineInfo.pWaitSemaphoreValues = &waitValue;
timelineInfo.signalSemaphoreValueCount = 1;
timelineInfo.pSignalSemaphoreValues = &signalValue;

VkSubmitInfo submitInfo;
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.pNext = &timelineInfo;
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &timelineSemaphore;
submitInfo.signalSemaphoreCount  = 1;
submitInfo.pSignalSemaphores = &timelineSemaphore;
submitInfo.commandBufferCount = 0;
submitInfo.pCommandBuffers = 0;

vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
```

Additionally, timeline semaphores can be signaled directly from the host:

```
VkSemaphoreSignalInfo signalInfo;
signalInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO;
signalInfo.pNext = NULL;
signalInfo.semaphore = timelineSemaphore;
signalInfo.value = 2;

vkSignalSemaphore(dev, &signalInfo);
```

And like VkFence, they can be waited on directly from a host thread:

```
const uint64_t waitValue = 1;

VkSemaphoreWaitInfo waitInfo;
waitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
waitInfo.pNext = NULL;
waitInfo.flags = 0;
waitInfo.semaphoreCount = 1;
waitInfo.pSemaphores = &timelineSemaphore;
waitInfo.pValues = &waitValue;

vkWaitSemaphores(dev, &waitInfo, UINT64_MAX);
```

Their current value may also be queried directly from a host thread:

```
uint64_t value;
vkGetSemaphoreCounterValue(dev, timelineSemaphore, &value);
```

enabling applications to incorporate timeline semaphores in both busy wait and blocking wait algorithms.

Putting Timeline Semaphores To Work

With the new API in hand, let’s look at a complete example workflow using many of the new features exposed by timeline semaphores:

```
#include <thread>
#include <vulkan/vulkan_core.h>

// A single Vulkan device object.
extern VkDevice dev;

// Three independent Vulkan queues from VkDevice <dev>.
extern VkQueue queue1;
extern VkQueue queue2;
extern VkQueue queue3;

// One timeline semaphore object.
VkSemaphore timeline;

static void thread1()
{
  const uint64_t waitValue1 = 0; // No-op wait. Value is always >= 0.
  const uint64_t signalValue1 = 5; // Unblock thread2's CPU work.

  VkTimelineSemaphoreSubmitInfo timelineInfo1;
  timelineInfo1.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
  timelineInfo1.pNext = NULL;
  timelineInfo1.waitSemaphoreValueCount = 1;
  timelineInfo1.pWaitSemaphoreValues = &waitValue1;
  timelineInfo1.signalSemaphoreValueCount = 1;
  timelineInfo1.pSignalSemaphoreValues = &signalValue1;

  VkSubmitInfo info1;
  info1.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  info1.pNext = &timelineInfo1;
  info1.waitSemaphoreCount = 1;
  info1.pWaitSemaphores = &timeline;
  info1.signalSemaphoreCount  = 1;
  info1.pSignalSemaphores = &timeline;
  // ... Enqueue initial device work here.
  info1.commandBufferCount = 0;
  info1.pCommandBuffers = 0;

  vkQueueSubmit(queue1, 1, &info1, VK_NULL_HANDLE);
}
```

```
static void thread2()
{
  // Wait for thread1's device work to complete.
  const uint64_t waitValue2 = 4;

  VkSemaphoreWaitInfo waitInfo;
  waitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
  waitInfo.pNext = NULL;
  waitInfo.flags = 0;
  waitInfo.semaphoreCount = 1;
  waitInfo.pSemaphores = &timeline;
  waitInfo.pValues = &waitValue2;

  vkWaitSemaphores(dev, &waitInfo, UINT64_MAX);

  // ... Perform some CPU work dependent on thread1's device work here.

  // Unblock thread3's device work.
  VkSemaphoreSignalInfo signalInfo;
  signalInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO;
  signalInfo.pNext = NULL;
  signalInfo.semaphore = timeline;
  signalInfo.value = 7;

  vkSignalSemaphore(dev, &signalInfo);
}

static void thread3()
{
  const uint64_t waitValue3 = 7; // Wait for thread2's CPU work to complete.
  const uint64_t signalValue3 = 8; // Signal completion of all work.

  VkTimelineSemaphoreSubmitInfo timelineInfo3;
  timelineInfo3.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
  timelineInfo3.pNext = NULL;
  timelineInfo3.waitSemaphoreValueCount = 1;
  timelineInfo3.pWaitSemaphoreValues = &waitValue3;
  timelineInfo3.signalSemaphoreValueCount = 1;
  timelineInfo3.pSignalSemaphoreValues = &signalValue3;
  VkSubmitInfo info3;
  info3.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  info3.pNext = &timelineInfo3;
  info3.waitSemaphoreCount = 1;
  info3.pWaitSemaphores = &timeline;
  info3.signalSemaphoreCount  = 1;
  info3.pSignalSemaphores = &timeline;
  // ... Enqueue device work dependent on thread2's CPU work here.
  info3.commandBufferCount = 0;
  info3.pCommandBuffers = 0;

  vkQueueSubmit(queue3, 1, &info3, VK_NULL_HANDLE);
}

int main()
{
  // Create the timeline semaphore object
  VkSemaphoreTypeCreateInfo timelineCreateInfo;
  timelineCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO;
  timelineCreateInfo.pNext = NULL;
  timelineCreateInfo.semaphoreType = VK_SEMAPHORE_TYPE_TIMELINE;
  timelineCreateInfo.initialValue = 0;

  VkSemaphoreCreateInfo createInfo;
  createInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
  createInfo.pNext = &timelineCreateInfo;
  createInfo.flags = 0;

  vkCreateSemaphore(dev, &createInfo, NULL, &timeline);

  // Spawn three free-running CPU threads using the timeline semaphore
  std::thread t1(thread1);
  std::thread t2(thread2);
  std::thread t3(thread3);

  // Wait for the device and CPU work using the timeline semaphore to idle
  const uint64_t waitValue = 8;

  VkSemaphoreWaitInfo waitInfo;
  waitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
  waitInfo.pNext = NULL;
  waitInfo.flags = 0;
  waitInfo.semaphoreCount = 1;
  waitInfo.pSemaphores = &timeline;
  waitInfo.pValues = &waitValue;

  vkWaitSemaphores(dev, &waitInfo, UINT64_MAX);

  // Destroy the timeline semaphore object
  vkDestroySemaphore(dev, timeline, NULL);

  // Clean up the CPU threads.
  t3.join();
  t2.join();
  t1.join();

  return 0;
}
```

Note the three threads spawned in this example run in an arbitrary order on the host. This lack of host-side synchronization is made possible by the wait-before-signal support in timeline semaphores: Regardless of submission order, the behavior of the timeline semaphores, and hence the order of the device work, is well defined.

Additionally, this example contains CPU work in thread2() which is interlocked on both sides by device work submitted in thread1() and thread3(), demonstrating the device->host and host->device synchronization capabilities of timeline semaphores. As is demonstrated here, care must be taken to ensure the state of the semaphore is well defined when signaling it from the host. While both the host and device signal operations are atomic, there is no implicit ordering guarantee relative to other host or device signal operations. That, combined with the monotonicity requirement, means applications must ensure any prior or concurrently submitted device signal operations are explicitly ordered relative to host signals, as is the case here due to the prior host wait and subsequent device wait. Further, if multiple threads or processes are to execute host signal operations on a given timeline semaphore, the application must ensure their execution proceeds in a similarly well-defined order to ensure monotonic value updates.

The example also demonstrates the flexible nature of the timeline semaphore state. While the application must ensure the value increases monotonically (that is, the value must never decrease), it is otherwise arbitrary, allowing it to serve a purpose beyond simple synchronization. For example, applications may find it useful to track a monotonic timestamp value using the semaphore state.

Finally, note the use of the timeline semaphore itself to determine when it is safe to destroy the timeline semaphore object. Prior to Vulkan 1.2, a separate VkFence object would have been required to perform such a task.

While timeline semaphores often remove the need for host-side synchronization of Vulkan device work submission, there is one shortcoming that many developers will run into when utilizing them: Vulkan’s window system integration APIs do not yet support timeline semaphores, and the wait-before-signal behavior of timeline semaphores is not inherited by binary semaphore objects. To illustrate the full impact of this limitation, let’s take a look at the above example code again, but with a presentation step incorporated. The additional code required is shown in red below:

```
#include <thread>
#include <atomic>
#include <vulkan/vulkan_core.h>

// A single Vulkan device object.
extern VkDevice dev;

// Three independent Vulkan queues from VkDevice <dev>.
extern VkQueue queue1;
extern VkQueue queue2;
extern VkQueue queue3;
extern VkQueue queue4;

// Swapchain data
extern VkSwapchainKHR swapchain;
extern uint32_t swapchainImageIndex;

// One timeline semaphore object.
VkSemaphore timeline;

// One binary semaphore object for vkQueuePresent()
VkSemaphore binary;

// Track signal submission count.
std::atomic_uint64_t submitCount(0);

static void thread1()
{
  const uint64_t waitValue1 = 0; // No-op wait. Value is always >= 0.
  const uint64_t signalValue1 = 5; // Unblock thread2's CPU work.

  VkTimelineSemaphoreSubmitInfo timelineInfo1;
  timelineInfo1.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
  timelineInfo1.pNext = NULL;
  timelineInfo1.waitSemaphoreValueCount = 1;
  timelineInfo1.pWaitSemaphoreValues = &waitValue1;
  timelineInfo1.signalSemaphoreValueCount = 1;
  timelineInfo1.pSignalSemaphoreValues = &signalValue1;

  VkSubmitInfo info1;
  info1.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  info1.pNext = &timelineInfo1;
  info1.waitSemaphoreCount = 1;
  info1.pWaitSemaphores = &timeline;
  info1.signalSemaphoreCount  = 1;
  info1.pSignalSemaphores = &timeline;
  // ... Enqueue initial device work here.
  info1.commandBufferCount = 0;
  info1.pCommandBuffers = 0;

  vkQueueSubmit(queue1, 1, &info1, VK_NULL_HANDLE);
  submitCount++;
}

static void thread2()
{
  // Wait for thread1's device work to complete.
  const uint64_t waitValue2 = 4;

  VkSemaphoreWaitInfo waitInfo;
  waitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
  waitInfo.pNext = NULL;
  waitInfo.flags = 0;
  waitInfo.semaphoreCount = 1;
  waitInfo.pSemaphores = &timeline;
  waitInfo.pValues = &waitValue2;

  vkWaitSemaphores(dev, &waitInfo, UINT64_MAX);

  // ... Perform some CPU work dependent on thread1's device work here.

  // Unblock thread3's device work.
  VkSemaphoreSignalInfo signalInfo;
  signalInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO;
  signalInfo.pNext = NULL;
  signalInfo.semaphore = timeline;
  signalInfo.value = 7;

  vkSignalSemaphore(dev, &signalInfo);
  submitCount++;
```

```
}

static void thread3()
{
  const uint64_t waitValue3 = 7; // Wait for thread2's CPU work to complete.
  const uint64_t signalValue3 = 8; // Signal completion of pre-present work.

  VkTimelineSemaphoreSubmitInfo timelineInfo3;
  timelineInfo3.sType = VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO;
  timelineInfo3.pNext = NULL;
  timelineInfo3.waitSemaphoreValueCount = 1;
  timelineInfo3.pWaitSemaphoreValues = &waitValue3;
  timelineInfo3.signalSemaphoreValueCount = 1;
  timelineInfo3.pSignalSemaphoreValues = &signalValue3;

  const VkSemaphore signalSemaphores[2] = {
    timeline, // Track timeline semaphore work completion
    binary // Unblock presentation
  };

  VkSubmitInfo info3;
  info3.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  info3.pNext = &timelineInfo3;
  info3.waitSemaphoreCount = 1;
  info3.pWaitSemaphores = &timeline;
  info3.signalSemaphoreCount  = 2;
  info3.pSignalSemaphores = signalSemaphores;
  // ... Enqueue device work dependent on thread2's CPU work here.
  info3.commandBufferCount = 0;
  info3.pCommandBuffers = 0;

  vkQueueSubmit(queue3, 1, &info3, VK_NULL_HANDLE);
  submitCount++;
}

int main()
{
  // Create the timeline semaphore object
  VkSemaphoreTypeCreateInfo timelineCreateInfo;
  timelineCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO;
  timelineCreateInfo.pNext = NULL;
  timelineCreateInfo.semaphoreType = VK_SEMAPHORE_TYPE_TIMELINE;
  timelineCreateInfo.initialValue = 0;

  VkSemaphoreCreateInfo createInfo;
  createInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
  createInfo.pNext = &timelineCreateInfo;
  createInfo.flags = 0;

  vkCreateSemaphore(dev, &createInfo, NULL, &timeline);

  // Create the binary semaphore object
  VkSemaphoreCreateInfo binaryCreateInfo;
  binaryCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
  binaryCreateInfo.pNext = NULL;
  binaryCreateInfo.flags = 0;

  vkCreateSemaphore(dev, &binaryCreateInfo, NULL, &binary);

  // Spawn three free-running CPU threads using the timeline semaphore
  std::thread t1(thread1);
  std::thread t2(thread2);
  std::thread t3(thread3);

  // Wait for submission of all dependencies of the binary semaphore
  while (submitCount < 3);

  // Present results
  VkPresentInfoKHR presentInfo;
  presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  presentInfo.pNext = NULL;
  presentInfo.waitSemaphoreCount = 1;
  presentInfo.pWaitSemaphores = &binary;
  presentInfo.swapchainCount = 1;
  presentInfo.pSwapchains = &swapchain;
  presentInfo.pImageIndices = &swapchainImageIndex;
  presentInfo.pResults = NULL;
  vkQueuePresentKHR(queue4, &presentInfo);

  // Wait for the device and CPU work using the timeline semaphore to idle
```

```
    const uint64_t waitValue = 8;

    VkSemaphoreWaitInfo waitInfo;
    waitInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO;
    waitInfo.pNext = NULL;
    waitInfo.flags = 0;
    waitInfo.semaphoreCount = 1;
    waitInfo.pSemaphores = &timeline;
    waitInfo.pValues = &waitValue;

    vkWaitSemaphores(dev, &waitInfo, UINT64_MAX);

    // Destroy the timeline semaphore object
    vkDestroySemaphore(dev, timeline, NULL);

    // Clean up the CPU threads.
    t3.join();
    t2.join();
    t1.join();

    return 0;
}
```

Note the application is now required to ensure **all** its work has been submitted before submitting the presentation work which waits on a binary semaphore. This is because the binary semaphore wait requires not only that its corresponding signal be submitted, but also that any work potentially blocking that signal operation be submitted. Hence, mixing binary and timeline semaphores is discouraged unless necessary, as it is above.

Clever readers may wonder why a second timeline semaphore object rather than a C++11 atomic was not utilized here to track work submission on the host side. This is to illustrate another point: While timeline semaphore signals and waits are themselves atomic operations, there is no increment or decrement operation exposed by the timeline semaphore API. Hence, utilizing a timeline semaphore would have imposed otherwise unnecessary execution ordering constraints on the host threads. Another alternative would have been to submit the presentation request only after the host wait operation in the main thread. While this would preserve the free-running nature of the host worker threads, it would introduce a new serialization point between the main host thread and the device, reducing overall parallelism.

While some inconvenient limitations in the API remain, the timeline semaphore programming model should greatly reduce the need for host-side synchronization and the number of synchronization objects Vulkan applications are required to track, thereby reducing host-side stalls and application complexity, which should in turn increase performance and quality. As such, the Vulkan working group highly encourages all developers to make the switch to timeline semaphores for all coarse-grained synchronization purposes. To help facilitate this switch while acknowledging the need to support older Vulkan implementations in the field, an implementation of the timeline semaphore API as a Vulkan 1.1 layer has also been provided under the permissive Apache 2.0 license as part of the Vulkan-ExtensionLayer (https://github.com/KhronosGroup/Vulkan-ExtensionLayer/) project.

Applications may incorporate this code directly into their projects, ship it as a layer alongside their applications, or utilize it in any other way to emulate the Vulkan 1.2 timeline semaphore API on older implementations as necessary, allowing them to program to a single synchronization API in all environments.

A higher-level introduction to the features and limitations of timeline semaphores is also available in slide and video form, as presented at the SIGGRAPH 2019 Khronos Birds of a Feather sessions:

- SIGGRAPH 2019 Khronos BOF Timeline Semaphore Presentation Recording (https://www.youtube.com/watch?v=1fU4w2ZGxH4&t=1800s)
- SIGGRAPH 2019 Khronos BOF Timeline Semaphore Slides (https://www.khronos.org/assets/uploads/developers/library/2019-siggraph/Vulkan-04-Timeline-Semaphore-SIGGRAPH-Jul19.pdf)

Authored by James Jones, NVIDIA, Vulkan Working Group