API Collection

Resource synchronization

Prevent multiple commands that can access the same resources simultaneously by coordinating those accesses with barriers, fences, or events.

Overview

You need to synchronize the commands you submit to an `MTL4CommandQueue` instance that have conflicting resource accesses. An *access conflict* is when multiple commands can access the same resource at the same time, and at least one of those accesses is a write operation. Access conflicts can cause problems in your app, such as nondeterministic behavior. For example, one command that reads from a resource might start before another command finishes writing its data to the same resource on which that the first commands depends.

First, identify which resources' accesses are in conflict, and then address the conflict with one of the following synchronization mechanisms.

> **Important**
>
> The value of an `MTLResource` instance's `hazardTrackingMode` property has no effect on the work you submit to an `MTL4CommandQueue`.

Identify memory access aliasing

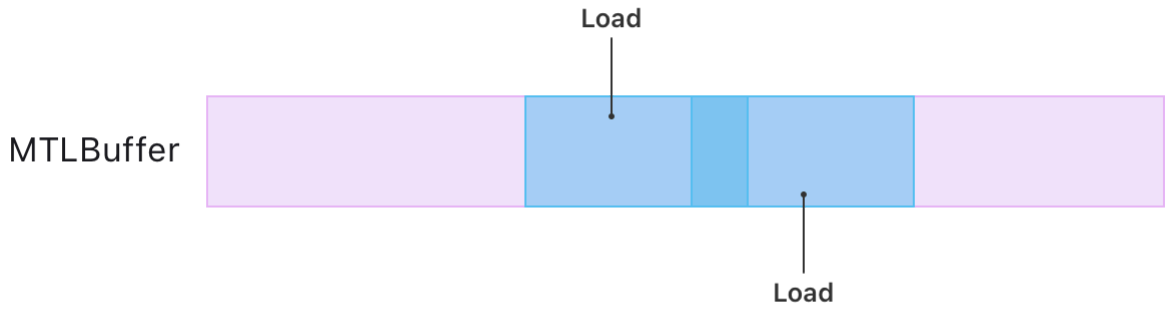
Start by finding which resources you need to synchronize by identifying the commands that have conflicting memory accesses. Every *access* to a Metal resource, including buffers and textures, is either a load or a store operation:

Load operation
: Reads data from a resource's underlying memory, such as sampling a texture.

Store operation
: Writes data to the resource's memory. Memory *aliasing* is when your app encodes two or more commands that access the same underlying memory at the same time.

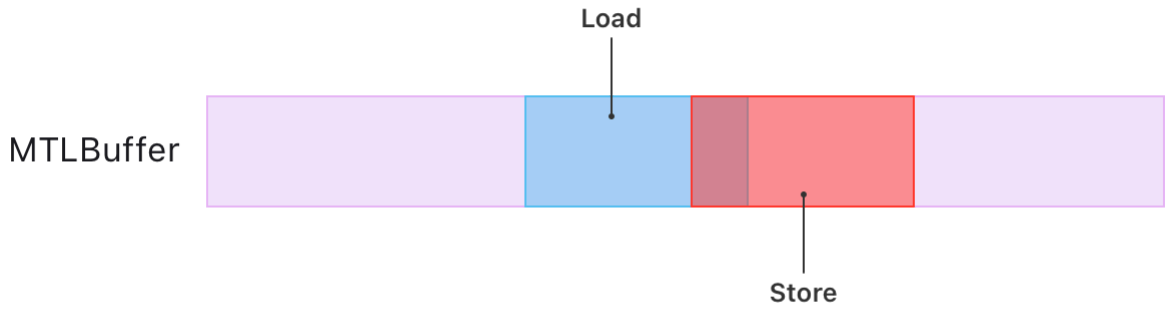
Aliasing isn't a problem if all the commands only read from the communal resource with load operations because none of the commands modify the resource's underlying memory by writing to it with a store operation.

For example, multiple commands can load segments of the same resource at the same time, even if those segments overlap, because none of them are writing to that memory.



Identify potential memory access conflicts

An app can introduce an access *conflict* when it encodes commands that both read and write to the same resource. A GPU typically runs multiple commands at the same time — by design, and each access conflict that can run concurrently creates a race condition because the overlapping memory load and store accesses don't always run in the same order relative to each other. For example, encoding commands that both load from and store to the same bytes of an `MTLBuffer` instance creates an access conflict.



The load operation from one command might run before, after, or at the same time as the other command's store operation because a GPU can run these commands in parallel.

> **Note**
>
> Even though a race condition is typically the result of an implementation mistake, some apps intentionally introduce race conditions as an optimization technique, such as running two identical resource stores that have consistent results that don't need synchronization.

Locate potential access conflicts by checking which resources apply to multiple commands, where at least one of those commands modifies the resource with a store operation. Consider resources that multiple encoders can access through any means, including argument buffers and resource bindings, which you configure directly to an encoder or with an argument table. Access conflicts can also apply to a render pass's color, depth, and stencil texture attachments.

> **Important**
>
> Apps that encode a render pass that accesses an attachment may introduce an access conflict because a render encoder can implicitly add load and store operations for those attachments.

Render command encoders add a load operation, a store operation, or both for each applicable texture attachment of the render pass it encodes. The attachment load operations run at the beginning of the render pass and the attachment store operations run at the end of it.

You configure which attachments, if any, the GPU loads at the beginning of the pass by setting the `loadAction` property of the `MTLRenderPassAttachmentDescriptor` instance that applies

to that attachment to `MTLLoadAction.load`. Separately, you configure which attachments, if any, the GPU stores stores at the end of the pass by setting the `storeAction` property of the `MTLRenderPassAttachmentDescriptor` instance that applies to that attachment to `MTLStoreAction.store`.

> **Note**
>
> You can use any combination of load and store actions for each attachment.

Also consider commands from a single encoder that can run concurrently. For example, the Metal 4 type `MTL4ComputeCommandEncoder` encodes a pass that only runs its commands concurrently on the GPU, which means any two commands that access the same memory can be a potential access conflict. One of the equivalent types, `MTLComputeCommandEncoder`, encodes a pass that runs its command serially by default, but you can configure it to encode a concurrent compute pass by setting an `MTLComputePassDescriptor` instance's `dispatchType` property to `MTLDispatchType.concurrent`.

Skip the accesses-the-system-already guarantees

Metal provides several built-in resource ordering guarantees within compute and render passes, which your app doesn't need to synchronize.

For example, you don't need to synchronize compute or render passes when they access an instance of an atomic type because they serially access those instances. See section 2.6 *Atomic Data Types* in the Metal Shading Language Specification (PDF) for more information.

Render passes also enforce access ordering for specific access types, including:

- Accesses to render-pass attachments run in primitive order for each fragment, which is the order of your app's draw commands and the order of each primitive within a draw call.
- Raster-order group accesses from a fragment shader run in primitive order for each fragment.
- Tile shader accesses to resources.
- Accesses to tile memory, which run in the order of your app's draw commands and tile-shader dispatch calls within the same tile.

Resolve conflicting accesses with synchronization

You can address a race condition from a memory access conflict by synchronizing those accesses with an appropriate mechanism. Synchronization ensures that a store operation completely finishes writing all updates to a resource's underlying memory before another command loads that the same resource, including the recent updates to its memory. Each synchronization mechanism

forces the GPU to pause before it runs a stage that accesses a resource, until another stage finishes.

You can choose one of the following synchronization mechanisms, which are in order of increasing scope:

Intrapass barriers
: An intrapass barrier has the smallest scope because it only applies to commands within the same pass. For more information, see Synchronizing resource accesses within a single pass with an intrapass barrier.

Fences
: A fence synchronizes access to resources across different passes within a command queue. You can select a specific stage within a producing pass that updates a fence, and a specific stage for a consuming pass to wait for a fence. For more information, see Synchronizing resource accesses between multiple passes with a fence.

Intraqueue barriers
: An intraqueue barrier also synchronizes access to resources across different passes within a command queue, but is more coarse than a fence. Metal has two kinds of intraqueue that synchronize different dependencies

  - A consumer queue barrier indicates which stages of the current pass depend on and consume output from specific stages of one or more previous passes in the same queue. For more information, see Synchronizing resource accesses with earlier passes with a consumer-based queue barrier.
  - A producer queue barrier indicates which stages of the current pass produce output that subsequent passes on the same queue depend on. For more information, see Synchronizing resource accesses with subsequent passes with a producer-based queue barrier.

Events
: An `MTLEvent` instance synchronizes resource accesses in passes across all command queues.

Shared events
: An `MTLSharedEvent` instance has the largest scope because it synchronizes resource accesses across:

  - The app's main code that runs on the CPU
  - Passes in any command queue for the current `MTLDevice` instance
  - Passes in any command queue for other `MTLDevice` instances

> **Tip**
>
> Select the synchronizing mechanism with smallest scope that can resolve the access conflict because larger scopes pause the GPU from doing more work than smaller scopes.

Track hazards with the framework prior to Metal 4

The Metal framework automatically synchronizes resource access conflicts for the commands you submit to an `MTLCommandQueue` instance, and only for the resources that:

- You configure its `hazardTrackingMode` property to `MTLHazardTrackingMode.tracked`
- You directly bind to an encoder type that conforms to `MTLCommandEncoder`

Resources you create from an `MTLDevice` instance default to `MTLHazardTrackingMode.tracked`, and the resources you create from an `MTLHeap` instance default to `MTLHazardTrackingMode.untracked`. For more information, see Resource fundamentals and Memory heaps.

Topics

Synchronization

Use semaphores or events to coordinate actions across threads to avoid multi-threaded resource contention by copying shared data to multiple buffers.

- Synchronizing resource accesses within a single pass with an intrapass barrier

  Resolve resource access conflicts between stages within a single pass by adding an intrapass barrier.

- Synchronizing resource accesses between multiple passes with a fence

  Resolve resource access conflicts between multiple passes within a single command queue by signaling a fence in one pass and waiting for it in another.

- Synchronizing resource accesses with earlier passes with a consumer-based queue barrier

  Resolve resource access conflicts between multiple passes within a single command queue by creating a consumer-based intraqueue barrier.

- Synchronizing resource accesses with subsequent passes with a producer-based queue barrier

  Resolve resource access conflicts between multiple passes within a single command queue by creating a producer-based intraqueue barrier.

{} Synchronizing CPU and GPU work

Avoid stalls between CPU and GPU work by using multiple instances of a resource.

{} Implementing a multistage image filter using heaps and fences

Use fences to synchronize access to resources allocated on a heap.

`struct MTLStages`

Describes stages of GPU work.

`protocol MTLFence`

A memory fence to capture, track, and manage resource dependencies across command encoders.

`struct MTLRenderStages`

The stages in a render pass that triggers a synchronization command.

`struct MTLBarrierScope`

Describes the types of resources that a barrier operates on.

`struct MTL4VisibilityOptions`

Memory consistency options for synchronization commands.

Signal events

Events synchronize resource data across multiple command buffers, GPU devices, or processes.

{} Implementing a multistage image filter using heaps and events

Use events to synchronize access to resources allocated on a heap.

About synchronization events

Synchronize access to resources in your app by signaling events.

Synchronizing events within a single device

Use nonshareable events to synchronize your app's work within a single device.

Synchronizing events across multiple devices or processes

Use shareable events to synchronize your app's work across multiple devices or processes.

Synchronizing events between a GPU and the CPU

Use shareable events to synchronize your app's work between a GPU and the CPU.

`protocol MTLEvent`

A simple semaphore to synchronize access to Metal resources.

`protocol` `MTLSharedEvent`

An instance you use to synchronize access to Metal resources across multiple CPUs, GPUs, and processes.

`class` `MTLSharedEventHandle`

An instance you use to recreate a shareable event.

`class` `MTLSharedEventListener`

A listener for shareable event notifications.

`typealias` `MTLSharedEventNotificationBlock`

A block of code invoked after a shareable event's signal value equals or exceeds a given value.

See Also

Resources

Resource fundamentals

Control the common attributes of all Metal memory resources, including buffers and textures, and how to configure their underlying memory.

Buffers

Create and manage untyped data your app uses to exchange information with its shader functions.

Textures

Create and manage typed data your app uses to exchange information with its shader functions.

Memory heaps

Take control of your app's GPU memory management by creating a large memory allocation for various buffers, textures, and other resources.

Resource loading

Load assets in your games and apps quickly by running a dedicated input/output queue alongside your GPU tasks.