API Collection

Resource fundamentals

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

Overview

A *resource* is a memory asset, such as an `MTLBuffer` or `MTLTexture`, that a GPU can access (see Buffers and Textures).

You can either allocate a resource from an `MTLDevice` instance or an `MTLHeap` instance (see Memory heaps). Metal sets a resource's `hazardTrackingMode` property to `MTLHazardTrackingMode.default` if you don't select another tracking mode. The default value depends on what Metal instance creates the resource.

Important

The value of an `MTLResource` instance's `hazardTrackingMode` property has no effect on the work you submit to an `MTL4CommandQueue` (see Resource synchronization) or resources that commands access through an argument buffer.

Each resource your app creates typically uses one of these storage modes:

`MTLStorageMode.private`
: Apps can only access resources in private storage from the GPU.

`MTLStorageMode.shared`
: Apps can access resources in shared storage from both the CPU and the GPU.

`MTLStorageMode.managed`
: Apps can access resources in managed storage from both the CPU and the GPU, just like shared storage. However, the GPU backs resources in managed mode with memory in private

storage.

Private mode resources give your app optimization opportunities that shared mode resources don't. Managed mode resources also give your app the same opportunities and allow your to app access them from the CPU.

Topics

Resource management

- Setting resource storage modes
  Set a storage mode that defines the memory location and access permissions of a resource.
- Choosing a resource storage mode for Apple GPUs
  Select an appropriate storage mode for your textures and buffers on Apple GPUs.
- Choosing a resource storage mode for Intel and AMD GPUs
  Select an appropriate storage mode for your textures and buffers on AMD and Intel GPUs.
- Copying data to a private resource
  Use a blit command encoder to copy buffer or texture data to a private resource.
- Synchronizing a managed resource in macOS
  Manually synchronize memory for a Metal resource in apps.
- Transferring data between connected GPUs
  Use high-speed connections between GPUs to transfer data quickly.
- Reducing the memory footprint of Metal apps
  Learn best practices for using memory efficiently in iOS and tvOS.

Residency sets

- Simplifying GPU resource management with residency sets
  Organize your resources into groups and influence when they become accessible to the GPU.

`protocol` `MTLResidencySet`
: A collection of resource allocations that can move in and out of resident memory.

`class` `MTLResidencySetDescriptor`
: A configuration that customizes the behavior for a residency set.

View pools

`protocol` **MTLResourceViewPool**

Contains views over resources of a specific type, and allows you to manage those views.

`class` **MTLResourceViewPoolDescriptor**

Provides parameters for creating a resource view pool.

`protocol` **MTLTextureViewPool**

A pool of lightweight texture views.

`class` **MTLTextureViewDescriptor**

Tensors

`protocol` **MTLTensor**

A resource representing a multi-dimensional array that you can use with machine learning workloads.

`class` **MTLTensorDescriptor**

A configuration type for creating new tensor instances.

`class` **MTLTensorExtents**

An array of length matching the rank, holding the dimensions of a tensor.

`class` **MTLTensorReferenceType**

An object that represents a tensor in the shading language in a struct or array.

`struct` **MTLTensorUsage**

The type that represents the different contexts for a tensor.

`let` **MTLTensorDomain**`: String`

An error domain for errors that pertain to creating a tensor.

`protocol` **MTLTensorBinding**

An object that represents a tensor bound to a graphics or compute function or a machine learning function.

`struct` **MTLTensorError**

`enum` **Code**

The error codes that Metal can raise when you create a tensor.

`enum MTLTensorDataType`

The possible data types for the elements of a tensor.

`let MTLTensorDomain: String`

An error domain for errors that pertain to creating a tensor.

`var MTL_TENSOR_MAX_RANK: Int32`

Sparse resources

`enum MTLBufferSparseTier`

Enumerates the different support levels for sparse buffers.

`struct MTL4CopySparseBufferMappingOperation`

Groups together arguments for an operation to copy a sparse buffer mapping.

`struct MTL4UpdateSparseBufferMappingOperation`

Groups together arguments for an operation to update a sparse buffer mapping.

`enum MTLTextureSparseTier`

Enumerates the different support levels for sparse textures.

`struct MTL4CopySparseTextureMappingOperation`

Groups together arguments for an operation to copy a sparse texture mapping.

`struct MTL4UpdateSparseTextureMappingOperation`

Groups together arguments for an operation to update a sparse texture mapping.

Common resource functionality

`typealias MTLGPUAddress`

A 64-bit unsigned integer type appropriate for storing GPU addresses.

`protocol MTLAllocation`

A memory allocation from a Metal GPU device, such as a memory heap, texture, or data buffer.

`protocol MTLResource`

An allocation of memory accessible to a GPU.

`struct` `MTLResourceOptions`

Optional arguments used to set the behavior of a resource.

`struct` `MTLResourceUsage`

Options that describe how a graphics or compute function uses an argument buffer's resource.

`struct` `MTLResourceID`

See Also

Resources

- 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.

- Resource synchronization

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