

[Metal](#) / Compute passes

API Collection

# Compute passes

Encode a compute pass that runs computations in parallel on a thread grid, processing and manipulating Metal resource data on multiple cores of a GPU.

## Overview

Your app can perform large-scale computation or prepare data for a subsequent GPU pass by encoding a compute pass that works on Metal resources in parallel. Compute passes are the part of your Metal pipeline meant for heavy parallelization of tasks requiring fast math, such as ray tracing.

Encode commands for your compute pass by creating an [`MTLCommandBuffer`](#) and using it to create a new compute command encoder, using a method like [`makeComputeCommandEncoder\(\)`](#) or [`makeComputeCommandEncoder\(descriptor:\)`](#). Add individual dispatches for functions and their data to the compute pass by calling the command encoder's methods. At the end of assigning data and dispatching a function call to the encoder, create a command that runs in your compute pass with [`endEncoding\(\)`](#).

### Note

Everything used to set up your compute pass is CPU thread-safe, except for [`MTLComputeCommandEncoder`](#). Synchronize [`MTLResource`](#) instances you share between the CPU and GPU with an [`MTLFence`](#), an [`MTLEvent`](#), or a completion callback.

For information on dispatching commands to encode, see the [`MTLComputeCommandEncoder`](#) reference. Compute passes also support indirect command buffers; for more information, see Dispatching from Indirect Command Buffers.

The following two samples demonstrate basic compute passes:

- See [Performing calculations on a GPU](#) for an example of configuring and running a compute pass that performs basic parallel math.
- See [Processing a texture in a compute function](#) for an example of using a compute pass to modify data for a render pass.

## Kernel arguments and argument tables

Compute commands that execute your code on GPU call *kernel functions* in your Metal shader, annotated with `[[ kernel ]]`. Each kernel has associated argument tables, such as the buffer argument table `[[ buffer(n) ]]`, used to access data associated with kernel arguments. In addition to annotations describing any argument table, some kernel arguments need information on their address space. For more information, see the following sections of the [Metal Shading Language Specification \(PDF\)](#):

- For compute kernels, Section 5.1.3
- For function argument tables, Section 5.2
- For address spaces, Section 4

In addition, kernels also use a function table to take advantage of function pointers, allowing them to call visible and intersection functions. Visible functions allow you to use function pointers in kernels, letting you use function stitching and link against Metal dynamic libraries at runtime. Ray tracers use intersection functions on [MTLAccelerationStructure](#) instances to perform quick intersection checks.

For more information on function stitching, dynamic libraries, and ray tracing, see:

- [Customizing shaders using function pointers and stitching](#)
- [Creating a Metal dynamic library](#)
- [Ray tracing with acceleration structures](#)

For information on per-architecture support for function tables in compute passes and other restrictions, see the [Metal feature set tables \(PDF\)](#).

## Argument buffers and memory residency

Compute kernels can access argument data to populate a Metal structure, using an [MTLBuffer](#) created by an [MTLArgumentEncoder](#). For an in-depth discussion of argument buffers, see [Improving CPU performance by using argument buffers](#). Using a resource in an argument buffer requires that it's resident in GPU memory for the duration of the pass. For a resource to be resident, allocate it with either the [MTLStorageMode.shared](#) or [MTLStorageMode.managed](#) mode.

Resources become resident on a per-instance basis by calling methods like `useResource(_ : usage:)` and heaps become resident by calling methods like `useHeap(_ :)`.

### Important

For the duration of your compute pass, don't access any resident resources on the CPU. Doing so in your app can cause GPU memory corruption, such as visual artifacts.

When using resident resources, avoid data corruption by using an appropriate [MTLHazardTrackingMode](#) or by manually managing memory barriers and fences for untracked resources with the methods in Synchronizing Across Command Execution.

## Using tile memory in a compute pass

Apple family GPUs offer fast, integrated graphics memory called *tile memory* that's shared between subsequent passes for fast access to data. Compute passes can reserve this memory space for threadgroup memory or imageblock memory, giving your compute functions the ability to access temporary data at low latency across your shaders.

For more information see the following sections of the [Metal Shading Language Specification \(PDF\)](#):

- Section 4.4 for information on the `threadgroup` memory space
- Section 4.5 for information on the `threadgroup_imageblock` memory space
- Section 2.11 for information on imageblocks
- Section 5.6 for information on `imageblock` attributes

Because tile memory resides on GPU only, you reserve memory on a tile block rather than copy data to it. Use the methods in Encoding Tile Memory Usage to prepare the appropriate block of memory for your kernel.

For device support and other tile memory limitations, see [Metal feature set tables \(PDF\)](#).

## Topics

### Essentials

- { } Performing calculations on a GPU

Use Metal to find GPUs and perform calculations on them.

{ } Processing a texture in a compute function

Create textures by running copy and dispatch commands in a compute pass on a GPU.

## Encoding a compute pass

Encode commands to prepare data and run a kernel in a compute pass.

📄 Creating threads and threadgroups

Learn how Metal organizes compute-processing workloads.

📄 Calculating threadgroup and grid sizes

Calculate the optimum sizes for threadgroups and grids when dispatching compute-processing workloads.

`protocol MTL4ComputeCommandEncoder`

Encodes a compute pass and other memory operations into a command buffer.

`protocol MTLComputeCommandEncoder`

An interface for dispatching commands to encode in a compute pass.

## Configuring a compute pipeline state

Define the GPU state for a kernel function call in your compute pass.

`class MTL4ComputePipelineDescriptor`

Describes a compute pipeline state.

`class MTLComputePipelineDescriptor`

An instance describing the desired GPU state for a kernel call in a compute pass.

`protocol MTLComputePipelineState`

An interface that represents a GPU pipeline configuration for running kernels in a compute pass.

`class MTLStageInputOutputDescriptor`

A description of the input and output data of a function.

`class MTLPipelineBufferDescriptor`

The mutability options for a buffer that a render or compute pipeline uses.

`class MTLPipelineBufferDescriptorArray`

An array of pipeline buffer descriptors.

```
struct MTLPipelineOption
```

Options that determine how Metal prepares the pipeline.

## Configuring a compute pass

Define how kernel functions get called throughout your compute pass, and any indirect command buffers to access.

```
class MTLComputePassDescriptor
```

A description of how to dispatch execution of pass commands and GPU performance sampling.

```
enum MTLDISPATCHTYPE
```

The type of dispatch method to use when calling encoded functions.

```
struct MTLDISPATCHTHREADGROUPSINDIRECTARGUMENTS
```

The data layout required for arguments needed to specify the size of threadgroups.

```
class MTLComputePassSampleBufferAttachmentDescriptor
```

A configuration that instructs the GPU where to store counter data from the beginning and end of a compute pass.

```
class MTLComputePassSampleBufferAttachmentDescriptorArray
```

A container that stores an array of sample buffer attachments for a compute pass.

---

## See Also

### Command encoders

- Render passes

Encode a render pass to draw graphics into an image.

- Machine-learning passes

Add machine-learning model inference to your Metal app's GPU workflow.

- Blit passes

Encode a block information transfer pass to adjust and copy data to and from GPU resources, such as buffers and textures.

### ☰ Indirect command encoding

Store draw commands in Metal buffers and run them at a later time on the GPU, either once or repeatedly.

### ☰ Ray tracing with acceleration structures

Build a representation of your scene's geometry using triangles and bounding volumes to quickly trace rays through the scene.